
Automated Validation of Minimum Risk Model-Based System Designs of Complex Avionics Systems

Dissertation

zur Erlangung des akademischen Grades Doktoringenieur
(Dr.-Ing.)

vorgelegt der Fakultät für Informatik und Automatisierung
der Technischen Universität Ilmenau

von

Dipl.-Inf. Nils Fischer
geboren am 19.02.1981 in Gotha

Gutachter: (1) Prof. Dr. Horst Salzwedel, TU Ilmenau
(2) Prof. Dr.-Ing. habil. Armin Zimmermann, TU Ilmenau
(3) Prof. Dr.-Ing. Mark Wiegmann, HAW Hamburg

Tag der Einreichung: 04.11.2016
Tag der wissenschaftlichen Aussprache: 24.10.2017

urn:nbn:de:gbv:ilm1-2017000432

Acknowledgements

Scientific research in a world with an exponentially growing amount of knowledge and ever-increasing system complexity is both challenging and exciting. This doctoral research is the result of my work for different research projects in the fields of model- and simulation-based systems engineering and aviation. My research would not have been possible without the guidance of many others.

First of all, I want to thank my parents for paving the way for my education and work and my wife, Barbara, who has always believed in me. Without you, this work would not have been possible.

I would like to express my special appreciation to my advisor Professor Dr. Horst Salzwedel, who encouraged my research in the first place and who kindled my interest in interdisciplinary research and philosophy. I am very grateful to Professor Dr. Armin Zimmermann who was always available for advice concerning my research and for agreeing to serve as the second examiner of this thesis. My very special thanks go to Professor Dr. Mark Wiegmann for his constant support, our many fruitful conversations and for joining my dissertation committee.

I further want to thank all my former colleagues at the Cabin and Cabin Systems Lab at the Department of Automotive and Aeronautical Engineering of the Hamburg University of Applied Sciences. I would also like to thank Dipl.-Inf. Tino Jungebloud and the company Mission Level Design GmbH for constant technical support in terms of modeling and simulation. I am very grateful for the long-standing collaboration with Airbus Group and for the possibility to work for a variety of research projects. Special thanks go to Nicholas Brownjohn for proofreading my thesis as well as to Stefan Osternack and Stéphane Poulain for their support within all our joint research projects at Airbus. Finally, my sincere thanks go to all my friends for all the years of patience and to all the people that I forgot to mention.

This research was partly funded under the fourth "*Luftfahrtforschungsprogramm*" (LuFo IV, Engl. federal aeronautic / aviation research program) of the German *Federal Ministry for Economic Affairs and Energy* (formerly known as *Federal Ministry of Economics and Technology*) under grant numbers *20K0702A*, *20K0806A*, *20K0805A* and *20K0805R*. Moreover, this research was sponsored by the German *Federal Ministry of Education and Research* under grant number *01IS13031A* as part of the research project "*Automatisierung der Architekturoptimierung komplexer Systeme*" (ARKOSE, Engl. automation of the architecture optimization of complex systems). The responsibility for the content of this publication lies with the author.

Supported by:



Federal Ministry
for Economic Affairs
and Energy

on the basis of a decision
by the German Bundestag

SPONSORED BY THE



Federal Ministry
of Education
and Research

Kurzfassung

Große zivile Flugzeuge umfassen eine hohe Anzahl von komplexen und gekoppelten Untersystemen mit Tausenden von elektronischen Steuergeräten und Software mit Millionen von Codezeilen. Flugzeughersteller sind bestrebt überlegene Produkte anzubieten, die mit minimalem Zeit- und Kostenaufwand bei maximaler Sicherheit entwickelt werden. Die komplexen Wechselwirkungen eines solchen Systems aus Systemen können von Einzelpersonen nicht vollständig verstanden werden. Vor allem die Suche nach einer optimalen Lösung gestaltet sich als unmöglich, in einer gewaltigen Menge von verschiedenen möglichen Systementwürfen, wenn diese manuell durchgeführt wird. Daher beinhalten geschriebene, nicht ausführbare Spezifikationen einen hohen Grad an Produktunsicherheit. Infolgedessen müssen mehr als zwei Drittel aller Spezifikationen überarbeitet werden. Da die meisten Spezifikationsfehler zu einem späten Zeitpunkt entdeckt und gelöst werden, wenn Aufwände für Überarbeitungen maximal sind, hat der gegenwärtige Entwicklungsansatz eine hohe Wahrscheinlichkeit für Kosten- und Zeitüberschreitungen oder führt zum Fehlschlagen von Projekten. Hierdurch wird das Entwicklungsrisiko maximiert.

Es ist das Ziel dieser Arbeit, eine modell- und simulationsbasierte Systementwurfsmethode mit zugehöriger Entwurfs- und Validierungsumgebung zu entwickeln, welche das Risiko der Entwicklung für komplexe Systeme minimiert, zum Beispiel für die Entwicklung von Flugzeugen. Das Entwicklungsrisiko ist minimal, wenn alle Entwicklungsentscheidungen frühzeitig vom Endkunden gegen die Leistungen eines Produktes auf Missionsebene validiert werden. Dazu werden ausführbare Spezifikationen während des Entwurfs erstellt und anhand der Anforderungen auf Missionsebene validiert. Validierte ausführbare Spezifikationen werden für alle Entscheidungen von der Konzeptentwicklung bis zur Implementierung verwendet und aktualisiert. Darüber hinaus werden virtuelle Prototypen entwickelt, welche ausführbare Systemspezifikationen mit Konzeptmodellen für die Mensch-Maschine-Schnittstellen kombinieren, um Usability-Anforderungen in den Gesamtentwurf aufzunehmen. Dies ermöglicht eine interaktive Spezifikationsvalidierung sowie frühes Endbenutzertraining mittels benutzergesteuerter Systemsimulation.

In einem ersten Schritt werden ausführbare Arbeitsabläufe und Simulation Sets entwickelt, welche die Ausführung von strukturierten und gekoppelten Simulationsmodellen ermöglichen. Anschließend wird ein modell- und simulationsbasiertes Entwicklungs- und Validierungsprozessmodell vom Konzeptdesign bis zur Spezifikationsentwicklung entwickelt. Hierfür werden zwei verschiedene Validierungsprozesse eingeführt. Ein automatisierter Validierungsprozess basierend auf ausführbaren Spezifikationen und ein interaktiver Validierungsprozess basierend auf virtuellen Prototypen. Für die Entwicklung von ausführbaren Spezifikationen und virtuellen Prototypen werden vorgefertigte Modellkomponenten spezifiziert. Die entwickelte Methode wird mit Hilfe von Beispielen aus der zivilen Flugzeugentwicklung validiert, insbesondere für die Entwicklung komplexer Avionik sowie für hochkonfigurierbare und individualisierbare Kabinensysteme.

Abstract

Today, large civil aircraft incorporate a vast array of complex and coupled subsystems with thousands of electronic control units and software with millions of lines of code. Aircraft suppliers are challenged to provide superior products that are developed at a minimum time and cost, with maximum safety and security. No single person can understand the complex interactions of such a system of systems. Finding an optimal solution from large sets of different possible designs is an impossible task if done manually. Thus, written, non-executable specifications carry a high degree of product uncertainty. As a result, more than two-thirds of all specifications need to be reworked. Since most specification flaws are discovered and resolved at a late stage during development, when expenditures for redesign are at a maximum, the development approach currently used has a high probability of project cost and time overruns or even project failure, thus maximizing the risk of development.

It is the aim of this work, to develop a model- and simulation-based systems engineering methodology with associated design and validation environment that minimizes the risk of development for complex systems, e.g. aircraft. The development risk is a minimum, if all development decisions are validated early against the services of a product at mission level by the final customer. To do so, executable specifications are created during design and validated against the requirements of system services at mission level. Validated executable specifications are used and updated for all decisions from concept development through implementation and training. In addition, virtual prototypes are developed. A virtual prototype is an executable system specification that is combined with human machine interface concept models to include usability requirements in the overall design and to enable interactive specification validation and early end user training by means of interactive user-driven system simulation.

In a first step, so called executable workflows and simulation sets are developed to enable the execution of sets of structured and coupled simulation models. Moreover, simulation sets are required to perform automated specification validation and system optimization. In a second step, a model- and simulation-based development and validation process model is developed from concept design to detailed specification development. In a final step, two different validation processes are developed. An automated validation process based on executable specifications and an interactive validation process based on virtual prototypes. For the development of executable requirements specifications, system specifications, and virtual prototypes, plug-and-play capable model components are developed. The developed methodology is validated for examples from civil aircraft development, especially in context of avionics and highly configurable and customizable cabin systems.

Contents

1	Introduction	1
1.1	Challenges of Aircraft Development	1
1.2	Contributions of This Thesis	6
1.3	Thesis Structure	8
2	Detailed Problem Analysis and Review of the State-of-the-Art	9
2.1	Background - Current Design and Development Life Cycle of Aircraft	9
2.2	Problem Analysis - Product Uncertainty and Risk of Development . .	16
2.3	Existing Solution Approaches	25
2.3.1	Developing the Right System	25
2.3.2	Designing Systems Correctly	26
2.3.3	Model-Based Systems Engineering: From Lilienthal Until Today	31
2.3.4	Detailed Analysis - Design at Mission Level	43
2.3.4.1	Excursion - Scenario-based Requirements Analysis .	44
2.3.4.2	Mission-level Design Approach Evaluation	46
2.3.4.3	The System Design Environment MLDesigner	50
2.3.5	Automation Approaches for MBSE	52
2.3.5.1	Validation Automation	52
2.3.5.2	Verification Automation	55
2.3.5.3	Test Automation	56
2.3.5.4	Automated Model Generation and Optimization . . .	57
2.3.5.5	Workflows	58
2.4	Detailed Objectives of This Work	59
2.4.1	Minimum Risk Model-Based Systems Engineering	59
2.4.2	Executable Workflows and Simulation Sets	61
2.4.3	MR-MBSE Design Environment	63
2.5	Summary	65
3	A Model-Based Design Method for Complex Avionics Systems	67
3.1	Overview	67
3.2	MR-MBSE Concept	70
3.2.1	Requirements Specification Model Development	70
3.2.2	Overall System Specification Model Development	72
3.3	Summary	76
4	Minimum Risk Model-based Systems Engineering Method	77
4.1	Executable Workflows for MR-MBSE	77
4.1.1	Control Flow and Control Nodes	80
4.1.2	Information Exchange and Data Flow	86

4.1.3	Towards Simulation Sets	87
4.2	Conceptual Design with MR-MBSE	89
4.2.1	Project Scope, System Context and Stakeholders	90
4.2.2	Top-Level Requirements Elicitation and Analysis	91
4.2.3	Concept Validation	95
4.3	Executable Requirements Specification Development	96
4.3.1	Overall Mission Model Development	100
4.3.2	Atomic Mission Model Development	105
4.3.2.1	Actor, Interaction and Data Model Development	108
4.3.2.2	Customization Model Development	118
4.3.3	Quality Objective Model Development	125
4.3.4	System and Environment Configuration Model Development	130
4.3.5	Overall Service Model Development	135
4.3.5.1	Service Model Development	137
4.3.5.2	Scenario Model Development	139
4.3.6	Human Machine Interface Concept Model Development	148
4.3.6.1	Graphical User Interface Model Development	150
4.3.6.2	Graphical User Interface Customization	152
4.3.6.3	Non-Functional Parameter Visualization	152
4.3.6.4	System and Environment Configuration	154
4.3.7	Requirements Documentation and Tracing	155
4.3.8	Executable Requirements Specification Validation	157
4.4	Executable Overall System Specification Development	161
4.4.1	System Specification Model Development	164
4.4.1.1	Function Model Development	168
4.4.1.2	Architecture Model Development	170
4.4.1.3	Environment Model Development	173
4.4.2	Automated Specification Validation	174
4.4.3	Towards Automated Reliability Analysis	180
4.4.4	Interactive Specification Validation, Test and Training	181
4.4.5	Overall System Design Optimization	187
4.5	Summary	191
5	Analysis and Validation of the Developed MR-MBSE Method	193
5.1	Design of a Basic Cabin Management System	193
5.2	CMS Scope, Context and Stakeholders	194
5.3	CMS Conceptual Design and Validation	195
5.4	CMS Executable Requirements Specification	199
5.4.1	CMS Mission Model Development	200
5.4.2	CMS Atomic Mission Model Development	203
5.4.3	CMS Quality Objective Model	212
5.4.4	CMS Environment Configuration Model Development	214
5.4.5	CMS Service Model Development	215
5.4.6	HMI Concept Model Development for CMS	224
5.4.7	CMS Requirements Specification Validation	226
5.5	CMS Executable Overall System Specification	227
5.5.1	Executable CMS Specification Model	227
5.5.2	Automated CMS Specification Validation	233
5.5.3	CMS Virtual Prototype	235

5.6	Avionics Systems Design Optimization	240
5.7	Summary	246
6	Concluding Remarks	247
	Bibliography	253
A	MR-MBSE Library	285
A.1	Top-Level Library	285
A.2	Simulation Set Library	287
A.3	Mission Model Library	289
A.4	Service Model Library	292
A.5	Validation Report Library	293
B	Basic Cabin Management System	295
B.1	Developed Atomic Mission Models	295
B.2	MLDesigner Scenario Flowchart PAX Call	304
	B.2.1 State Specification	304
	B.2.2 Transition Specification	311
B.3	Overall Data and Customization Model	315
B.4	Generated Validation Report	322
B.5	Generated Validation Report Tabular	329

List of Figures

1.1	Growing software complexity for aircraft	2
1.2	Seat availability graph example	3
1.3	Code size development of the CIDS main server	4
1.4	Global air traffic forecast	5
2.1	Structured development process for aircraft	10
2.2	Visualization of the SAE ARP 4754 development process	12
2.3	V-Model for aircraft development	13
2.4	Types of specifications during system development	15
2.5	Slotted system development process	18
2.6	Occurrence of design flaws during system development	20
2.7	Product design uncertainty and development costs	22
2.8	Committed life-cycle costs during development	23
2.9	The goals of successful system development	25
2.10	The beginnings of model-based aircraft development	32
2.11	Growth of avionics software and equipment	37
2.12	Overview of model-based system design levels	39
2.13	Examples and benefits for model-based development	41
2.14	Common model-based design process	43
2.15	Relation between use cases, scenarios and requirements specifications	46
2.16	System-level design approach	47
2.17	Simulation-based validation	47
2.18	Mission level design approach	48
2.19	Graphical user interface of MLDesigner	51
2.20	Hierarchical modeling principle of MLDesigner	52
2.21	Path analysis structure example	53
2.22	Automated software requirements specification validation process	54
2.23	Requirements and comparison sequence diagram mapping example	55
2.24	Scientific workflow example	58
2.25	Basic graphical simulation set composition example	62
2.26	Requirements for an integrated design and validation platform	65
3.1	Minimum risk model-based systems engineering principle	68
3.2	Iterative development cycle with executable specifications	69
3.3	MR-MBSE as part of the V-model development process	70
3.4	MR-MBSE flowchart - concept development	71
3.5	MR-MBSE flowchart - specification development	72
3.6	Virtual prototype development	73
3.7	From executable specification development to automated validation	74

3.8	Automated specification optimization	75
4.1	Executable workflow structure	79
4.2	Token-based control flow sequence	81
4.3	Initial, final and branch end control node	82
4.4	Split control node	82
4.5	Synchronized join control node	83
4.6	Unsynchronized join control node	83
4.7	TF-decision and CE-decision control node	84
4.8	Timed event and random timed event control node	85
4.9	Executable workflow example	86
4.10	Process handling primitives	87
4.11	External simulation control module	88
4.12	Concept design and validation process	90
4.13	Conceptual design mind map	95
4.14	Process for executable requirements specification development	98
4.15	Process for overall mission model development	100
4.16	Mission model and system under design	102
4.17	General mission model structure example	103
4.18	Atomic mission model structure	107
4.19	Relation between atomic mission model and system under design	110
4.20	Hierarchical data model of atomic mission models	111
4.21	Atomic mission models and executable overall system specification	113
4.22	Actor modules for atomic mission models	114
4.23	Actor modules for executable specifications	115
4.24	Link mechanism between actors	116
4.25	Composite data structure <i>ACTOR DS</i>	117
4.26	Hierarchical customization data model structure	119
4.27	Customization model development	121
4.28	Customization actor modules	124
4.29	Customization model exchange	125
4.30	Relation between quality objective model and non-functional observers	127
4.31	Quality objective and non-functional observer modules	129
4.32	System and system environment configuration parameter sets	131
4.33	System and system environment configuration modules	133
4.34	Process for overall service model development	136
4.35	Service model structure	138
4.36	Actor Data Exchange (ADE) module	139
4.37	Scenario model structure	142
4.38	Actor input/output and validation report modules	143
4.39	Scenario flowchart structure	145
4.40	From mission model development to HMI model development	149
4.41	Use of Tcl/Tk for graphical user interface model development	151
4.42	Quality objective visualization module	153
4.43	Graphical user interface model development	154
4.44	Extended conceptual design mind map	157
4.45	Validation flow for executable requirements specifications	158
4.46	Process for executable overall system specification development	162
4.47	Process for detailed system development and validation	163

4.48	Structure of executable system specification models	167
4.49	Structure of function models	169
4.50	Relation between functions and elements	171
4.51	Structure of element models	172
4.52	Executable validation flow structure	175
4.53	Validation process evaluation and report modules	177
4.54	Automatically created validation reports	179
4.55	Automated validation report comparison	180
4.56	Structure of a simulation set for automated reliability analysis	181
4.57	Interactive specification validation	183
4.58	Interactive validation report generation	185
4.59	Interactive validation process report and probe modules	187
4.60	Overall system specification optimization process	190
5.1	System context and stakeholder analysis example	195
5.2	Analysis of top-level requirements for a basic cabin management system	196
5.3	Essential use case diagram example for a cabin management systems	196
5.4	Conceptual design mind map example for a cabin management system	197
5.5	Misuse case diagram example for cabin management systems	198
5.6	Lower level use cases for a cabin management system	198
5.7	Sequence diagram refinement for a cabin management system	199
5.8	Mission model example for a cabin management system	202
5.9	Atomic mission and data model visualization for use case <i>PAX Call</i> .	204
5.10	Atomic mission model for use case <i>PAX Call</i>	205
5.11	Atomic mission model for use case <i>LAV Smoke</i>	206
5.12	Customization and updated data model for use case <i>PAX Call</i>	208
5.13	Customization and updated data model for use case <i>LAV Smoke</i>	209
5.14	Atomic mission model for an automated FSB indication service	211
5.15	Final atomic mission model example for use case <i>LAV Smoke</i>	212
5.16	Element example with non-functional observer modules	213
5.17	Custom system environment configuration statechart module	215
5.18	Service model example for atomic mission model <i>LAV Smoke</i>	216
5.19	Scenario flowchart example for atomic mission model <i>PAX Call</i>	217
5.20	User interface design concepts for a basic CMS	225
5.21	Top level of an executable overall system specification for a CMS	227
5.22	System level view of an executable specification for a CMS	229
5.23	Structure of elements, sub-elements and functions of a CMS	230
5.24	Formal design evaluation example	233
5.25	Executable validation flow for a basic cabin management system	234
5.26	GUI models of a virtual prototype for a CMS	236
5.27	Multi-purpose cabin attendant user interface	237
5.28	Cabin user interface examples for interactive CMS simulations	238
5.29	Aircraft system and environment configuration widget examples	239
5.30	Cabin management system mockup	239
5.31	Architecture model and aircraft topology	240
5.32	Architecture model generation process	241
5.33	Avionics optimization flow (top-level)	242
5.34	Avionics optimization model	242
5.35	Avionics system architecture evaluation model	243

5.36	Simulation results for avionics systems architecture optimization . . .	244
5.37	Joint evaluation of behavior model and architecture model execution	245
6.1	MR-MBSE for aircraft in relation to actors, methods and tools	249
6.2	Minimization of product uncertainty and development risk	251

List of Tables

4.1	System and system environment configuration file format	134
4.2	Validation data structure	176
4.3	Interactive validation report file format	184
4.4	Tabular format for executable interactive validation probe files	186
5.1	Quality objectives for a basic CMS	234
5.2	Scenario configuration overview for a basic CMS	234
5.3	Comparison of quality requirements and properties for a basic CMS	235
5.4	Simulation platform specifications	235
5.5	Comparison of two architecture optimization simulations	245

List of Abbreviations

AS_x	Actor Stimuli Sequence
C_M	Overall Development Cost per Month
ER_x	Expected SuD Response
L_S	Learning Effect After Product Delivery
U_D	Learning Curve for Products During Development
U_L	Product Uncertainty Emerging as part of the Product Lifecycle
U_P	Overall Product Uncertainty
A/C	Aircraft
AADL	Architecture Analysis and Design Language
AAIB	Air Accidents Investigation Branch
ACARE	Advisory Council for Aeronautics Research and Innovation in Europe
ACMS	Aircraft Condition Monitoring System
ACRE	Approach to Context-based Requirements Engineering
ACT	Actor
ADE	Actor Data Exchange
AFDX	Avionics Full-Duplex Switched Ethernet
AFWAL	Air Force Wright Aeronautical Laboratories
AI	Artificial Intelligence
Alf	Action Language for Foundational UML
AMC	Acceptable Means of Compliance
AMIS	Atomic Mission / Atomic Mission Model
AMRAAM	Advanced Medium Range Air-to-Air Missile
API	Application Programming Interface
APU	Auxiliary Power Unit
ARINC	Aeronautical Radio Incorporated
ARKOSE	Automatisierung der Architekturoptimierung komplexer Systeme
ARP	Aerospace Recommended Practice
ASI	Actor Signal Input
ASO	Actor Signal Output
ATA	Air Transport Association
ATM	Air Traffic Management System
AUTO	Automatic
AUV	Autonomous Underwater Vehicle
BMBF	German Federal Ministry of Education and Research
BMWi	German Federal Ministry for Economic Affairs and Energy
BONeS	Block-Oriented Network Simulator
BSS	Boeing Satellite Systems
C	Control Node / Channel
CAD	Computer Aided Design

CAE	Computer Aided Engineering
CASE	Computer-aided Software Engineering
CBD	Component-based Development
CCP	Constraint-relevant Customization Parameters
CCS	Cabin Core System / Cabin and Cabin Systems
CCS-Lab	...	Cabin and Cabin Systems Laboratory
CE	Condition/Else
CFD	Computational Fluid Dynamics
CIDS	Cabin Intercommunication Data System
CM	Customization Model
CMS	Cabin Management System
CPCS	Cabin Pressure Control System
CS25	Certification Specification 25
CSV	Character-Separated Values
CVS	Concurrent Versions System
DAL	Design Assurance Level
DARPA	Defense Advanced Research Projects Agency
DB	Database / Customization Database
DDF	Dynamic Data Flow
DE	Discrete Event
DES	Discrete Event Simulation
DEVS	Discrete Event System Specification
DIMA	Distributed Integrated Modular Avionics
DLR	Deutsches Zentrum für Luft- und Raumfahrt
DM	Data Model
DMA	Actor Data Model
DMB	Director Mainboard
DMU	Digital Mock-Up
DoD	U.S. Department of Defense
DoDAF	Department of Defense Architecture Framework
DoS	Denial of Service
DS	Data Structure
E	Element
EASA	European Aviation Safety Agency
ECSAM	Embedded Computer System Analysis and Modeling
ECU	Electronic Control Unit
EDA	Electronic Design Automation
EFCS	Electrical Flight Control Systems
EM	Event Control Module
ENGSystem	Engine System / Power Plant
ENV	Environment / Environment Model / Environment Configuration
EOF	Executable Optimization Flow
EOL	End of Life
EOSS	Executable Overall System Specification
ER	Entity–Relationship
ERS	Executable Requirements Specification
ES	Executable Specification
ESL	Electronic System Level
ESPITI	European Software Process Improvement Training Initiative

ESPRIT	European Strategic Program on Research in Information Technology
ESS	Executable System Specification
EU	European Union
EU-OPS	European Union Operational Requirements
EUROCAE	.	European Organization for Civil Aviation Equipment
EV	Event Element
EVF	Executable Validation Flow
EW	Executable Workflow
F	Functional / Function
FAA	Federal Aviation Administration
FCP	Functional Customization Parameter
FCS	Flight Control System
FEA	Finite Element Analysis
FHA	Functional Hazard Assessment
FMEA	Failure Mode and Effects Analysis
FSB	Fasten Seatbelt
FSM	Finite State Machine
FTA	Fault Tree Analysis
fUML	Foundational UML
GB	Gigabyte
GUI	Graphical User Interface
HIL	Hardware-in-the-Loop
HMI	Human-Machine Interface
HoV	Head-of-Version
HTML	Hypertext Markup Language
HW	Hardware
I	Interface
IAI	Israel Aerospace Industries
IC	Integrated Circuit
IDE	Integrated Design Environment
IDVP	Integrated Development & Validation Process
IFE	In-flight Entertainment
IMA	Integrated Modular Avionics / Integrated Modular Architecture
INCOSE	International Council on Systems Engineering
Intercom	...	Intercommunication / Intercommunication System
IPCC	Intergovernmental Panel on Climate Change
IT	Information Technology
IVP	Interactive Validation Probe
IVV	Independent Validation and Verification
JAA	Joint Aviation Authorities
JSD	Jackson Systems Development
JSON	JavaScript Object Notation
LAAE	Logical Actor Assignment Expression
LAV	Lavatory
LAV Smoke	.	Lavatory Smoke Indication
LCC	Life Cycle Costs
LCE	Logical Customization Expression
LGSys	Landing Gear System
LOC	Lines of Code

LRM	Line Replaceable Module
LRU	Line Replaceable Units
LuFo	Luftfahrtforschungsprogramm
MB	Megabyte
MBE	Model-based Engineering
MBSE	Model-based Systems Engineering
MCS	Monte-Carlo Simulation
MDE	Model-driven Engineering
MIS	Mission
MLD	Mission Level Design / MLDesigner
MLDesigner	Mission Level Designer
MLOC	Million Lines of Code
MMMA	Multi Mission Maritime Aircraft
MOD	Module / Composite Module
MR-MBSE	Minimum Risk Model-based Systems Engineering
MTBF	Mean Time Between Failures
MTTR	Mean Time to Repair
MUC	Misuse Case
NASA	National Aeronautics and Space Administration
NDIA	National Defense Industrial Association
NF	Non-Functional / Non-functional Observer
NFCP	Non-functional or Quality Customization Parameter
NFP	Non-functional (Design) Property
NIST	National Institute of Standards and Technology
NS	No Smoking
O	Objective / Object Node
OMG	Object Management Group
ORM	Object Role Modeling
OSATE	Open Source AADL Tool Environment
PA	Public Address / Passenger Address
PAX	Passenger
PAX Call	Passenger Calls for Assistance
PBD	Platform-based Design
PED	Personal Electronic Device
PFA	Problem Frames Approach
PNS	Passenger Notification System
PRM	Primitive
PS	Parameter Set
PSSA	Preliminary System Safety Assessment
PSU	Passenger Supply Unit / Passenger Service Unit
PTcl	Ptolemy Tcl Interpreter Code
PTS	Purchaser Technical Specification
QFD	Quality-Function-Deployment
QO	Quality Objective
QOM	Quality Objective Model
R	Risk of System Development
r(E)	Resource of an Element
R-NET	Requirements Network
RADC	Rome Air Development Center

RBE	Requirements-based Engineering
RBSE	Requirements-based Systems Engineering
RE	Requirements Engineering
RFC	Request for Comments
RTCA	Radio Technical Commission for Aeronautics
RTS	Return to Seat
S	Section
SAE	Society of Automotive Engineers
SAIC	Science Applications International Corporation
SATA	Serial AT Attachment
SBSE	Simulation-based Systems Engineering
SCADE	Safety Critical Application Development Environment
SCM	Scenario Model
SCT	System Control Technology
SDES	Stochastic Discrete Event Systems
SDF	Synchronous Data Flow
SEM	Service Model
SFC	Scenario Flowchart
SIM	Simulation / Simulation Model
SIMKAB	Simplifizierte Kabine (Engl. simplified cabin)
SimSet	Simulation Set
SoS	System-of-Systems
SPW	Signal Processing Work-System
SRD	System Requirements Document
SREM	Software Requirements Engineering Methodology
SREP	Software Requirements Engineering Program
SSS	System Supplier Specification
SuD	System under Design / System under Development
SuO	System under Optimization
SuT	System Under Test
SVN	Apache Subversion
SW	Software
SysML	Systems Modeling Language
SYSMOD	Systems Modeling Process / Systems Modeling Toolbox
Tcl	Tool Command Language
TF	True/False
Tk	Tool Kit
TL	Top Level
TMLLV	Terrain-masking Low-level Flight System
TOPCASED	Tk in Open Source for Critical Applications & Systems Development
TPT	Time Partition Testing
U	Uncertainty
U2TP	UML 2.0 Testing Profile
UC	Use Case
UML	Unified Modeling Language
UTP	UML Testing Profile
V&V	Validation and Verification
VAL	Validation
VEOSS	Validated Executable Overall System Specification

VER	Verification
VERS	Validated Executable Requirements Specification
VES	Validated Executable Specification
VP	Virtual Prototype
VR	Virtual Reality
WCETs	Worst-case Execution Times
XMI	XML Metadata Interchange
XML	Extensible Markup Language
xUML	Executable UML

1. Introduction

“Gradual development of flight should begin with the simplest apparatus and movements, and without time complication of dynamic means.”

— Otto Lilienthal, quoted by Charles C. Turner, *The Romance of Aeronautics: An Interesting Account of the Growth & Achievements of All Kinds of Aerial Craft* [352]

The above quotation by Otto Lilienthal dates back to the 19th century, when flying machines were at the brink of becoming available to the public. During this period, Otto Lilienthal became fascinated with flying. He dreamed of humans being able to fly in a free and unlimited way, to cross borders, to unite countries and reconcile people [201]. Back then, aircraft were humble machines with low complexity, designed to lift just a single person into the air. Today, at the beginning of the 21st century, aircraft have a much higher degree of system complexity. They are omnipresent and allow comfortable mass transportation of people and freight worldwide. This chapter explores the meaning of system complexity and the currently existing challenges during aircraft development. Moreover, it describes the motivation for this work as well as associated research questions and concludes with an overview of the structure of this work.

1.1 Challenges of Aircraft Development

Large-scale systems and System-of-Systems (SoS), e.g. aircraft, spacecraft and large distributed software systems have always been a major challenge for system developers worldwide. These systems are characterized by distributed system architectures with a significant large number of coupled and complex subsystems. Additionally, millions of adjustable parameters and entities exist within a dynamically changing environment that work together to form a whole [74], [211] and [307]. Large aircraft in particular challenge design teams in terms of space and resources, both strongly limited, and stringent standards for safety and reliability. All these efforts are taken into account to ensure an important mission: safe and reliable flight while providing maximum passenger comfort. The number of people needed to tackle the development of aircraft easily reaches more than 10.000 [304] and, in the case of the A380,

an additional 34.000 people working for subsystem suppliers [23]. Moreover, modern aircraft like the Airbus A380 consist of more than 120 different systems [12] and are produced by over 1.500 companies from over 30 countries around the world [362]. But even today, Crossley states [92], the complex interactions between subsystems of aircraft are only loosely understood.

The complexity of aircraft and avionics, i.e. aviation electronics [59], has been advancing rapidly with exponential growth [120], closely following Moore's Law [226] and [99]. This is due to a growing amount of functions and electronic capabilities to increase safety, autonomy, satisfy passenger demands and the desire to reduce resources, cost and energy usage. Strong market competition forces the reduction of development time as new competitors emerge and leads to a high degree of outsourcing [71], [168] and [14]. On the other hand, recent developments for complex space systems by *SpaceX*, a company that only uses a minimum amount of outsourcing, showed that both cost and risk for development can be reduced significantly, e.g. compared to traditional use of aerospace contractors by the National Aeronautics and Space Administration (NASA) [78]. However, major aircraft manufacturers put a strong focus on the decomposition of the overall aircraft into subsystems that are developed independently by different system suppliers [12]. Currently, the biggest contributions to the rising system complexity within aircraft and avionics (the entirety of electronic equipment fitted in an aircraft) are exponentially increasing requirements for signal processing and communications [99], as depicted by Figure 1.1. The number of networked electronic control units (ECUs) integrated within avionics systems for civil aircraft started to grow past 1000 with the start of this millennium [311] and, in the case of the Airbus A380, the currently largest available civil aircraft, reached a number of more than 5000 ECUs [314].

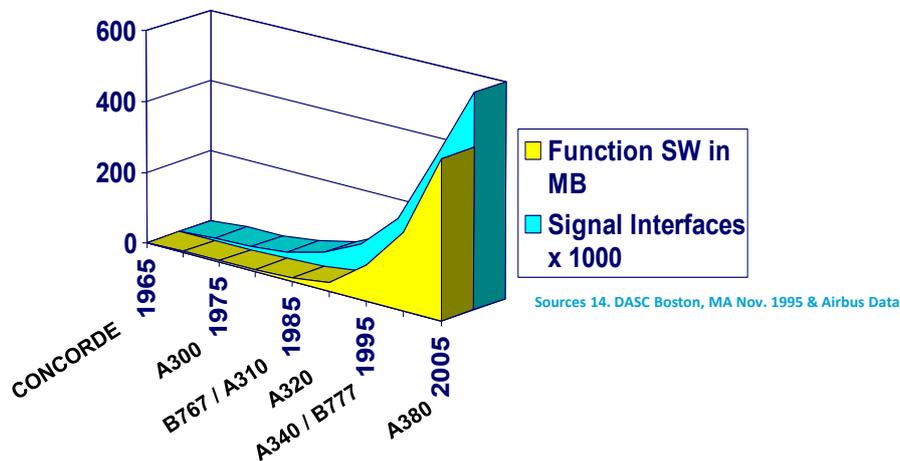


Figure 1.1 – Growing complexity in terms of signal processing and communications within avionics and cabin systems illustrated by the amount of functional software (SW) in megabyte (MB) and the number of signal interfaces (Source: Wiegmann 2012 [372], originally developed in reference [68])

In 1926, on a *Lufthansa* flight from Berlin to Frankfurt in Germany, an audio entertainment equipment of the *Telefunken and Ultraphon AG* was presented to the selected members of the public for the first time. The equipment provided music from a recorded concert as well as recitations from a novel during flight [206]. Today, air travel is shifting from a mere transportation oriented business into a business,

that provides passengers with a most attractive inflight experience [31]. Current inflight entertainment (IFE) systems for large commercial aircraft can contain well over 2000 interconnected line replaceable units (LRUs) [13] and [31]. New customer requirements also arise from an aging society and a growing interest to provide air travel for handicapped people [379]. As a result of this trend, a great amount of flexibility is required for each of the different cabin systems [381]. The growing importance of cabin systems for commercial aircraft can also be measured in terms of cost. While the most expensive subsystem for commercial aircraft is the propulsion system, the second most expensive system is the IFE system [372].

In the case of civil aircraft, cabin and related avionics systems have become the most complex and fastest changing systems in terms of hardware and software [287]. These systems only have a product life cycle of four to seven years, compared to 30 years for the rest of the aircraft [65]. A recent study notes, that taking advances in technology and expertise into account, the costs for development and certification of a major cabin system have increased more than fivefold, comparing the Airbus A320 (first flight in 1987) to the Airbus A350 (first flight in 2013) [183]. Especially on long-haul flights, passengers expect to find their rapidly changing comfort electronics technology known from consumer products like smart phones and tablet PCs. This also includes easy-to-use human-machine interfaces (HMI) that are harmonized within the cabin environment [287] and [126] as well as integrated personal electronic devices (PED) [266]. Hence, connectivity services for passengers, especially for wireless applications, have become a huge design driver, e.g. for IFE systems and other cabin systems [381]. IFE systems have come to provide nearly the same range of functionality as currently available consumer products [31].

Although not determined to be highly safety critical, cabin systems are essential for the success of airlines around the world because they are considered mission critical, i.e. contribute to the success or failure of an airline [374]. For major airlines, e.g. *Emirates*, the operability of the IFE system is a key factor for being able to perform a flight. If not operable, an aircraft might even stay on ground. Therefore, airlines closely monitor system performances to counter emerging problems as quickly as possible [13]. However, the margin for errors resulting from the unavailability of seats due to, e.g. the loss of an in-seat IFE component, is very small [358]. Figure 1.2 shows an example graph for typical seat availabilities, with a worst case of 99,8% for overall seat availability in August. Although this number may seem minuscule, it already is exceeding the tolerable limits for most airlines [13], [358], and [374].



Figure 1.2 – Seat availability graph example (Source: Virilli and Reitmann 2006 [358])

Another pivotal system for successful operation of many aircraft is the cabin management system (CMS). CMS changed substantially from federated system architectures with single switches for cabin systems control, e.g. cabin lighting and temperature, to integrated, networked, and highly configurable system architectures with multi-purpose user interfaces [293] and an increasing amount of software [65]. Today, a CMS provides all essential abilities, functions and services for safe and efficient operation of commercial aircraft [373]. Because of its pivotal role for a large set of cabin-related services, a CMS is also referred to as cabin core system (CCS) [160]. A good example for functional diversity of CMS for current large commercial aircraft is the *Cabin Intercommunication Data System* (CIDS), common to all current Airbus aircraft. More than 5000 single system requirements contribute to over 40 major system functions and services. Important services include passenger address (PA) and cabin announcement functions, cabin environment control, potable water and waste indication, emergency evacuation signaling, lavatory smoke detection, passenger service related functions as well as cabin and service interphone [287] and [289].

A typical Airbus aircraft equipped with CIDS, e.g. the Airbus A380, has more than 350.000 single system components with more than 600 LRUs and more than 600 printed circuit boards. The CIDS is digitally coupled with other aircraft systems and processes more than 10.000 different input/output signals with the aid of one of the largest on-board software systems, comprised of more than five million lines of code (MLOC) [289] and [65]. The software of the CIDS is mainly executed on a centralized, modular and integrated platform with shared resources [371]. Figure 1.3 shows the growth in code size for the CIDS server component from 2002 to 2012 [65].

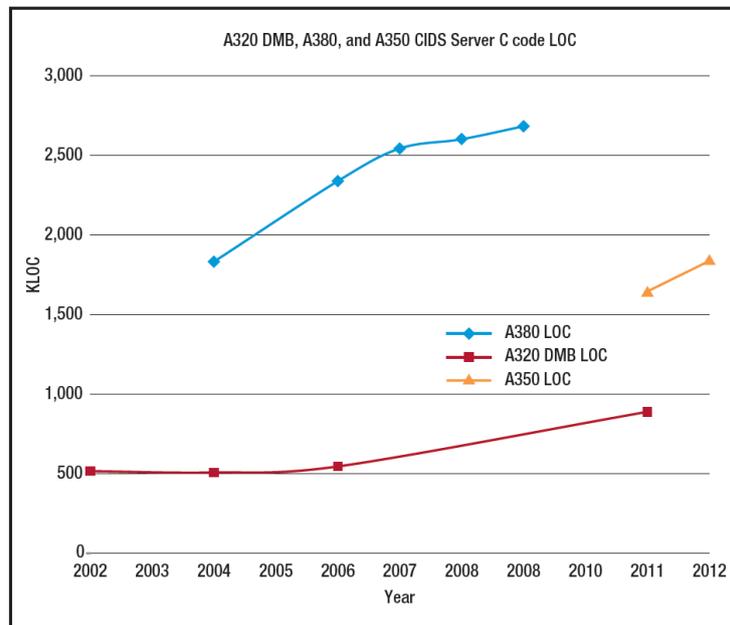


Figure 1.3 – Code size development in lines of code (LOC) of the CIDS main server software for Airbus aircraft during 2002 and 2012 (Source: Burger et al. 2013 [65])

In contrast to all other aircraft systems, cabin systems are strongly affected by configuration and customization processes. For each aircraft, airlines may choose number and configuration of seats, layouts, related electronic units, and functions. A key decision strongly affecting cabin services, underlying network, and supply systems [372] and [130]. In the case of the CIDS for example, approximately one

million adjustable system parameters are provided for system customization and configuration [289]. Systems customization is used to add, remove or alter functions, architecture components or user interfaces according customer expectations whereas system configuration is used for determining number, assignment and arrangement of system components together with performance parameter settings.

The first aircraft with a specific configuration and customization is called *Head-of-Version* (HoV). Each HoV needs to be validated, verified, tested, and certified before serial reproductions of the HoV can be manufactured. Therefore, each HoV requires a large amount of work and can cause distinct delays for aircraft delivery [296]. Each airline usually orders several HoVs in order to cover a wide variety of aircraft and cabin operations. As an example, Airbus delivered 434 aircraft across all aircraft families with a total of 100 HoVs in 2006 [73]. Since airlines have growing demands for cabin customization, almost every cabin can currently be considered unique [12]. In the case of CMS for large commercial aircraft, systems customization and configuration causes a large amount of work and can currently take up to 101 days for a single aircraft [289]. According to former senior vice president for cabin and cargo at Airbus, Rüdiger Fuchs [360], “*Cabin customization accounts for about 10% of the value of a commercial transport [...] In the case of the A380, with a list price of about \$260 million, that would mean a “huge” amount of money [...]*” [360].

Today, public authorities as well as major aircraft suppliers predict stable growth rates for air traffic within the next two decades [2] with estimated doubling of the global large commercial airplane fleet by 2030 (see Figure 1.4) [319] and [6].

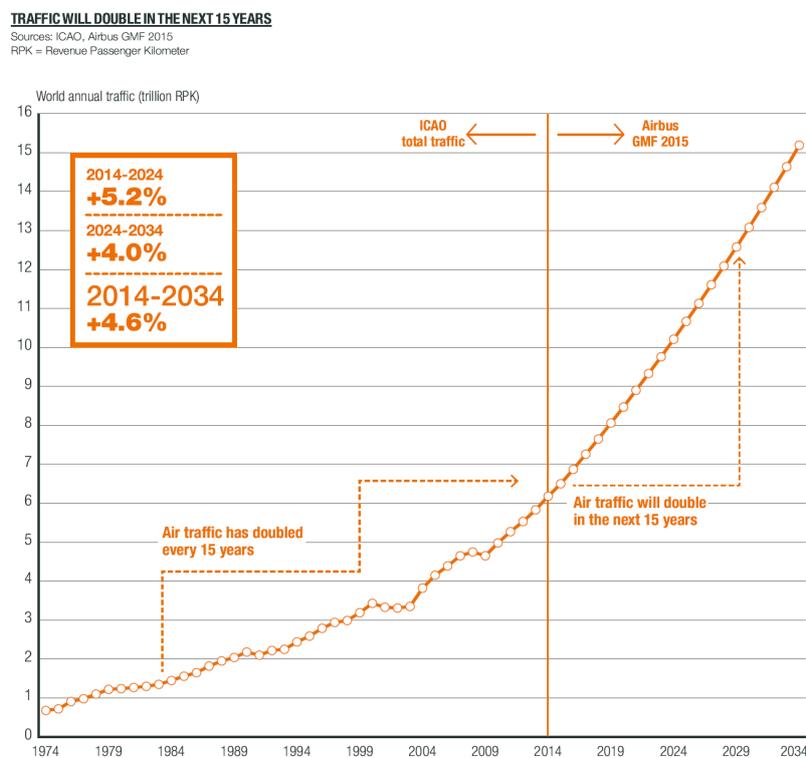


Figure 1.4 – Airbus air traffic forecast (Source: Airbus Global Market Forecast 2015 [319])

As a consequence thereof, cabin systems as part of the passenger service environment, will remain the major focus of attention by airlines as well as aircraft manufacturers [31]. Thus, there will be high demands for the development of superior

aircraft systems that are to be developed at minimum time, cost, weight, power consumption and maximum security.

Cabin system design is already driven by demands for the optimization of development time, weight, noise and overall cabin environment comfort [153] and [110]. Current cabin systems showed potential weight savings of about 30% with around 20% less installation place compared to previous systems [31]. The need for aircraft systems optimization is also driven by international organizations. The *Advisory Council for Aeronautics Research and Innovation in Europe* (ACARE) has determined a set of global requirements for the aviation sector within the next years, called *Vision 2020*. Principal objectives include a 50% reduction of carbon dioxide emissions and 80% nitrogen oxide emissions per passenger kilometer [19]. These requirements are also supported by analyses of the *Intergovernmental Panel on Climate Change* (IPCC) [291] as well as the *Greener by Design* initiative of the *Royal Aeronautical Society* [32]. Aircraft as well as aircraft subsystems of the future shall provide “[...] *cost-conscious travel with choice, comfort and convenience*” [19]. Therefore, “[...] *cost and efficiency of the aircraft as well as its design and manufacturing must be the most competitive in the world*” [19]. According to ACARE [19], an estimate funding of more than 100 billion Euro will be required to achieve all goals within the next 20 years [19]. Because aircraft subsystems and single components are already optimized, a major optimization for aircraft can only be achieved at aircraft level [316].

In summary, the development of aircraft is most complex and challenging in many ways, particularly in terms of overall design complexity, customization complexity, financial complexity and project management complexity [12]. Since aircraft development projects are extremely expensive and complex at the same time, they are prone to cost overruns or even project failure [12]. Thus, a high financial risk is associated with the development of aircraft. Recent developments such as the Airbus A380 or the Boeing B787 experienced massive delays and caused serious cost overruns. Overall development costs rose from more than \$5 billion U.S. dollar for the development of Boeing’s B777 during the early 1990s to about \$10 to \$11 billion Euro for the Airbus A380 in the first decade of 2000 [361] and [23]. One major reason was the number of design iterations needed due to a high number of changes within system specifications during development [95] and [215]. Hence, with the current design methodology, especially with regard to the bottom-up oriented development of systems, it is impossible to clearly understand the overall design or to optimize the overall system architecture.

1.2 Contributions of This Thesis

This thesis is motivated by the growing complexity of large dynamically coupled systems of systems, e.g. aircraft or spacecraft, developed by distributed teams and with high demands for safety, security, maturity and cost efficiency (cf. chapter 1). As elaborated within the previous chapter, the successful development of highly complex systems, e.g. aircraft, is a challenging task and, since the introduction of digital electronics, has led to massive cost overruns or complete project failures.

Since current designs for complex systems often use bottom-up development processes, complex interactions of the dynamically coupled overall system can neither

be understood nor validated and the overall system cannot be optimized for top-level requirements. Since this uncertainty is currently not resolved during early design stages, the majority of specifications need to be reworked as part of repetitive design iterations.

In the next chapters, recently failed or challenged system development projects are analyzed in relation to the currently used development process for complex systems and focuses on the development of large commercial aircraft. After the problem to be solved has been determined in detail, a detailed analysis of the state of the art and related work is performed in order to describe what has been done to decrease product uncertainty within system specifications (cf. chapter 2). Research questions in the context of this work include:

- What is the meaning of system complexity and dynamic coupling?
- What does it mean to be able to develop a system successfully?
- What is the importance of early design validation and optimization?
- How should the development process currently used be improved?
- How should the process of validation and optimization with regard to top-level requirements be improved?

This thesis contributes to the field of systems engineering and to the successful development of complex systems, e.g. aircraft. The aim of this thesis is to develop a top-down development and validation method with associated design and validation environments that enables the understanding and validation of the complex interactions of dynamically coupled systems of systems during early development stages. This is done to permit the development of disruptive complex products at minimum risk that are validated and optimized for top-level requirements during early design stages. Product uncertainty shall be minimized and the overall specification quality of complex systems shall be improved before detailed subsystem development. As a result of the application of the proposed method, the currently required high amount of late specification changes and design iterations shall be minimized (cf. chapter 3).

The proposed solution is developed based on the analysis and extension of existing solutions and related work in the field of model- and simulation-based engineering (cf. chapter 3) and applies conceptual design models, executable specification models and virtual prototypes as the key elements of the overall development process. To improve overall specification quality, this work proposes the early application of automated specification validation and automated optimization processes at overall system level. To permit the realization of the proposed development method, an integrated plug-and-play capable design and validation environment is developed (chapter 4).

The developed method and design environment are validated and evaluated for examples from civil aircraft systems development (cf. chapter 5).

1.3 Thesis Structure

Chapter 1 introduces current system design and development challenges for complex systems, i.e. large civil aircraft with focus on highly configurable and customizable cabin systems and avionics.

Chapter 2 provides background information for the currently used development process for aircraft and analyzes the problem to be solved in detail (*What is the root cause for current system development challenges?*). As part of this analysis, a detailed overview of the application of model-based system engineering (MBSE) methods in the history of aircraft development is provided (*What has been done?*). A more detailed analysis is provided for related work in context of the mission level design approach and MBSE approaches for the automation of design and validation activities. The chapter concludes with a summary of the analysis and proposes necessary improvements and detailed objectives for this work (*What needs to be done?*).

Chapter 3 determines the general concept for the proposed solution together with associated boundary conditions, based on the performed analysis of existing solutions and related work (*How will it be done?*). The chapter concludes with a summary.

Chapter 4 elaborates the developed methodology in detail. A detailed description is provided for all major steps of the system development process, including requirements elicitation and analysis, concept design phase, specification development and validation. Moreover, the developed components for a plug-and-play capable system design and validation environment are elaborated in detail.

Chapter 5 is used to demonstrate, validate and evaluate the developed methodology for examples from civil aircraft systems development.

Chapter 6 summarizes the contents and results of this work. Moreover, open issues and limitations of the developed methodology are discussed. The chapter concludes with perspectives for future enhancements and applications of the developed methodology.

2. Detailed Problem Analysis and Review of the State-of-the-Art

This chapter provides background information for the currently used development process for aircraft before the problem to be solved is introduced and analyzed in detail. Moreover, existing solution approaches and related work are analyzed and evaluated. A more detailed analysis is provided for related work within the field of model-based systems engineering (MBSE) and the mission level design approach. At the end of this analysis, it is described what has to be done to overcome the problem stated at the beginning of this chapter. The chapter concludes with a detailed description of the objectives and contributions of this work.

2.1 Background - Current Design and Development Life Cycle of Aircraft

The currently used development process for aircraft focuses on methodologies applying a document-centric, structured systems design approach with hierarchical systems decomposition [225] and [96], often referred to as *requirements engineering* (RE), *requirements-based engineering* (RBE) or *requirements-based systems engineering* (RBSE). This approach is widely accepted and has been extensively covered in the literature [39], [64], [349], [108] or [306].

Different general and aircraft specific quality and management standards are applied to design and development of civil aircraft. The most important standards comprise the DIN EN ISO 9000 family [231], DIN EN ISO 9001 [232] and DIN EN 9100 [234]. These standards call for a structured system design and development process, that is also recommended by the *European Organization for Civil Aviation Equipment* (EUROCAE) and the *Society of Automotive Engineers* (SAE International) within the *Aerospace Recommended Practice ARP4754 / ED-79A - Guidelines for Development of Civil Aircraft and Systems* [238], [138] and *ARP4761 - Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*. More information on the application of these standards can be found in reference [384]. Figure 2.1 depicts the set of structured and parallel processes that form the overall system development process for civil aircraft.

ARP4754 and ARP4761 are considered major guidelines for the development of aircraft and are recognized by aviation authorities such as the *Federal Aviation Administration* (FAA). Standards and recommendations are supplemented by development and certification guidelines for aircraft and related systems, which are supporting the certification process that every aircraft needs to complete, e.g. according to the *European Aviation Safety Agency* (EASA) provision *Certification Specifications 25 for Large Aeroplanes* (CS-25), including the *Acceptable Means of Compliance* (AMC) [106]. Important development guidelines include the design objectives *DO-254 / ED-80 - Design Assurance Guidance for Airborne Electronic Hardware* [274], [136] *DO-178B / DO-178C / ED-12B - Software Considerations in Airborne Systems and Equipment Certification* [273], [277] and [135], *DO-160G / ED-14G - Environmental Conditions and Test Procedures for Airborne Equipment* [275], [139] and *DO-333 / ED-216 - Formal Methods Supplement to DO-178C and DO-278A* [280], [140] published by the *Radio Technical Commission for Aeronautics* (RTCA Incorporated) in joint effort with EUROCAE as well as the *United States defense standard MIL-STD-882B - System Safety Program Requirement* [239] published by the *United States Department of Defense*.

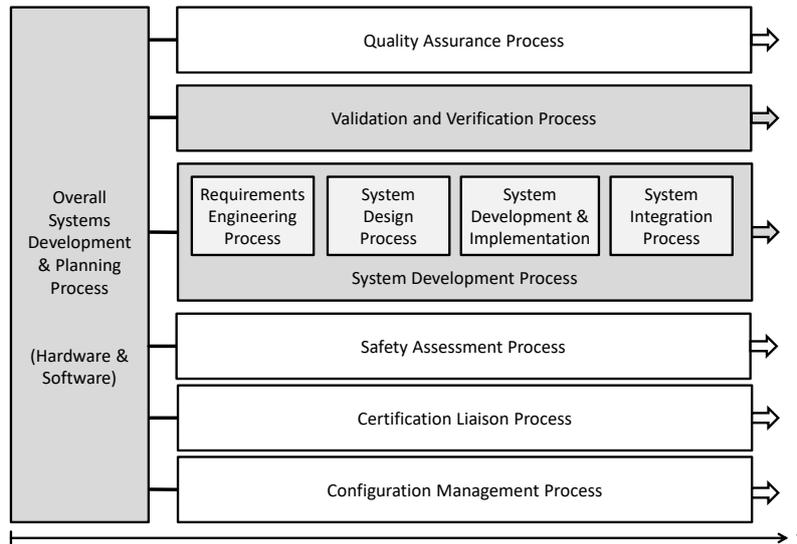


Figure 2.1 – Structured development process for aircraft with parallel processes for different development tasks as recommended by ARP4754 / ED-79A, ARP4761, CS-25, DO-178B / ED-12B and DO-254 / ED-80 (adapted from Berkhahn 2009 [36] and Wiegmann 2011 [371])

As shown in Figure 2.1, all sub processes for the development of aircraft are operated in parallel. In ISO 9000 and ISO 9100 standards, quality assurance includes all activities to secure that applicable processes, methods and tools are used correctly to ensure a certain degree of quality as expected by the standards [273]. The configuration management process comprises all activities that are used for maintaining the consistency of the product under development throughout the system life cycle. With respect to the requirements of the product, this includes intended product performance as well as functional and physical attributes, design and operational information [240], [166] and [274].

Unlike many other complex systems, aircraft development is accompanied by the highest demands for safety and reliability [33] due to the intention to transport hundreds of thousands of people via flight. Hence the process of certification liaison

is most important. Its foremost objectives are to establish communication and understanding between developers of aircraft and certification authorities such as the EASA or FAA throughout the system development cycle to assist with certification [273] and [274]. Airworthiness certification is mandatory by international law and includes all activities to comprehensively prove that a specific aircraft design fulfills all national and international safety regulations. To be able to prove safety requirements, a safety assessment process is carried out in contribution to certification liaison. Safety assessment is performed according to CS-25 subsection AMC 25.1309 and ARP4761 in conjunction with more specific manufacturer internal guidelines that are coordinated with certification authorities. Thus, for every aircraft development, an adequate level of confidence needs to be provided in order to substantiate, “[...] that errors in requirements, design, and implementation have been identified and corrected such that the system satisfies the applicable certification basis” (AMC 25.1309, sub-item i, Development Assurance, [106]). As depicted in Figure 2.2, system design information and intended aircraft function are used to perform a safety assessment for the overall system. The results of this concurrent process are passed on to system designers that will change their design accordingly. Methods used for safety assessment include, but are not limited to, *Preliminary System Safety Assessment* (PSSA), *Functional Hazard Assessment* (FHA), *Fault Tree Analysis* (FTA) as well as *Failure Mode and Effects Analysis* (FMEA) [52] and [220].

Airworthiness certification does not end with the development of one particular type of aircraft such as the Airbus A320 or Boeing B737. Certification is also mandatory during customization for every new aircraft configuration of a particular type of aircraft that has been defined and ordered by a customer, e.g. an airline. The first aircraft to represent a distinct aircraft configuration set for a specific type of aircraft is also referred to as *Head-of-Version* (HoV) [73]. More in-depth information on certification can be found in references [158] and [134].

Human machine interfaces (HMI) play an important role for safe and efficient operation of every aircraft and, in the case of cabin systems, need to provide a large set of functions for a heterogeneous user group, including passengers and crew. Experience shows, that current designs are not sufficient to provide an ideal user interface for all possible user groups. According to Manfred Sieber [328] at Airbus, it should be the aim of future cabin systems interface design, to unite requirements of system designers with the specific needs of users by means of a fitted system interaction concept [328]. Because of the importance of HMI development for future aircraft cabins, a common standard for cabin HMI specification development was developed [328] and [86]. Early assessment of mental models of user-system interaction, i.e. accessing the way users operate a system via corresponding interfaces, is one way to develop accurate interactive software [163]. To be able to design, validate and test human-machine interfaces (HMIs), early interactive testing with prototypes and direct involvement of future system users, e.g. by so called *Thinking Aloud Testing*, has been proven to be most reliable [328]. To achieve optimal use interface solutions, the application of virtual HMI prototyping is widely used in the aerospace industry [225].

Since more and more functions are integrated within aircraft that provide possible interfaces between aircraft systems, crew and passengers, the possibility of unwanted interaction between human or non-human actors and safety critical systems

is growing. Especially within the cabin domain, possible interfaces exist between e.g. inflight entertainment system and portable passenger devices through tethered or wireless connections. Moreover, different HMIs in the aircraft cabin might be operated by individuals other than the aircraft crew (unauthorized intrusion) [159]. Possible threats also include malware to infect systems and denial of service (DoS) attacks [276]. To complement other systems engineering methods, a security risk assessment process was created by the *National Institute of Standards and Technology* (NIST) to determine possible information security risks that act in opposition to the intended mission of a product [165]. In response to potential security related threads for aircraft, RTCA and EUROCAE recommend the introduction of an additional security process as vital certification qualification within *DO-326 / ED-202 - Airworthiness Security Process Specification* following the safety assessment [276] and [137].

The process of system development as well as the process of validation and verification will be elaborated more extensively within the next paragraphs, since it is essential for the problem statement in the context of this work. As depicted by Figure 2.2, two parallel sub development processes are executed for hard- and software that contribute to the overall system development process as defined by DO-254 and DO-178B. Because of their monolithic and autonomous structure, these applicable guidelines for hardware and software development for airborne systems do not cover the coupling between hardware (HW) and software (SW) early during the overall system development process [225].

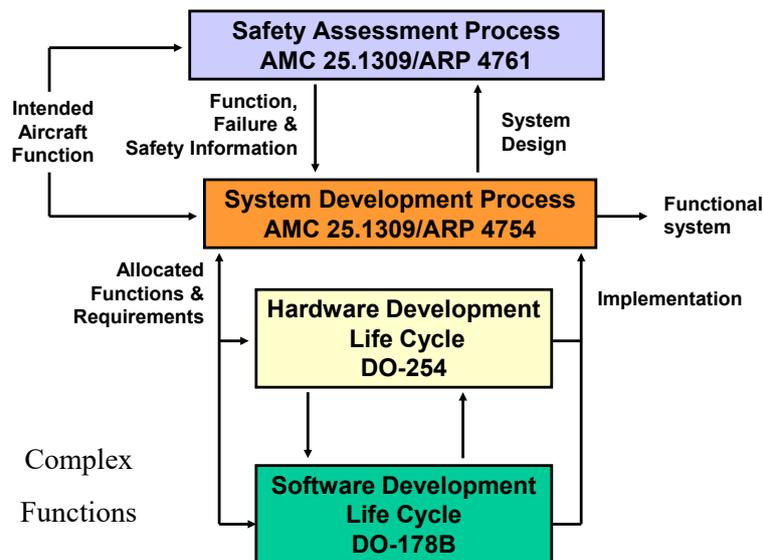


Figure 2.2 – Visualization of the SAE ARP 4754 development process. The design of hardware and software is not directly coupled during development and thus performed in parallel according to DO-178B and DO-254 (Source: Lee 2006 [193], based on [238])

The overall process methodology for planning and performing systems development projects for aircraft is based on the V-Model [354] with associated internal manufacturer guidelines [96] which reflect the above stated standards and guidelines [297] and [335]. The origins of the V-Model date back to the work on the development of software systems by Barry Boehm [46] and [48] during the late 1970s and early 1980s. In the following years, the V-Model was simultaneously developed in Germany [354] and the United States of America [141] for both, software and systems

engineering. Figure 2.3 depicts the currently used V-Model systems development process for aircraft that is identical for both, hard- and software. It consists of a top-down driven requirements engineering phase, an implementation phase and a bottom-up driven integration process. More specifically, the following steps are used: requirements elicitation influenced by safety and reliability assessment, design, decomposition and specification of the system that is to be developed (left-hand side), product development and implementation (beginning at the bottom) and system integration including test and rollout (right-hand side).

The distinction between all of the different design and development phases is not always clear, since no precise definition exists to distinguish carefully where one phase ends and another one begins. Moreover, multiple iterations over each phase are possible. These iterations are often regarded as part of a concurrent engineering process as encouraged by the *V-Model XT* [354] and [43].

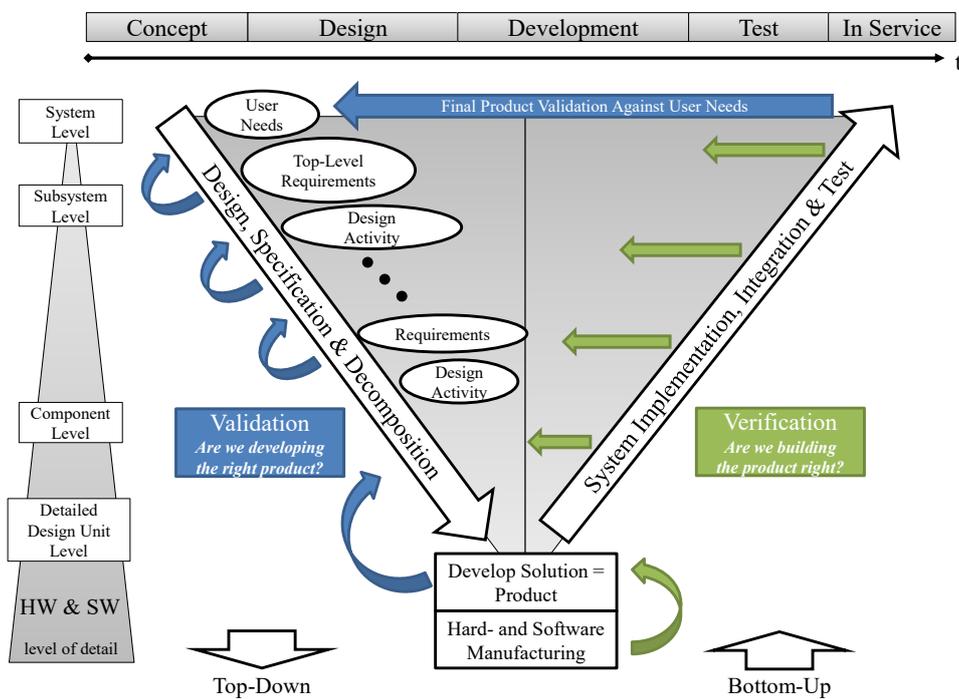


Figure 2.3 – V-Model for aircraft development (adapted from [354], [124], [297] and [335])

For every aircraft, the development process is starting with an idea, often derived from a certain need and a set of market analyses. The aircraft manufacturer derives top-level requirements for the product together with several stakeholders, i.e. airlines, marketing and system experts as well as certification authorities (via standards and guidelines). In this phase, distinct product characteristics and basic conditions are being analyzed and determined to specify the true needs for the system under design, i.e. the aircraft at overall system level. This is often referred to as concept design.

In the case of complex systems, the overall system may be characterized as, according to the International Council on Systems Engineering (INCOSE) [251], “[...] an interacting combination of elements to accomplish a defined objective. These include hardware, software, firmware, people, information, techniques, facilities, services, and other support elements.”. In the process of system design, product requirements

that are defined at different levels of detail may be generalized according to *Kotonya and Sommerville* [185] definition: “[...] requirements define the services that the system should provide and they set out constraints of the system’s operation.” [185]

Top-level as well as refined requirements specifications within the left-hand side of the V-Model include functional and non-functional requirements for the system under design (SuD). Functional requirements describe the capabilities, i.e. functions and behavior of a system as expected by the user of the system. Use-cases are one possibility that coherently describe specific parts of system functionality. The entirety of all use-cases forms the overall system behavior. Non-functional requirements describe necessary requisites for the applicability of the system. These include quality, performance, safety, reliability, political or commercial targets as well as constraints [354].

Thus, as a result of the concept design phase, written (top-level) requirements specifications, i.e. system requirements documents (SRD), are generated that document and comprise the fundamental requirements for the functional behavior of the system under design together with non-functional constraints and information about the interaction of the system with its environment [96]. Therefore, these specifications are the foundation for all further steps of the product design and development process and constitute a binding contract that is used by system suppliers during development [233].

After the design has been specified top-down and the given design task that has been decomposed and stated within requirements specifications, also referred to as purchaser technical specifications (PTS), sub-system and component suppliers (for hard- and software) start to develop a feasible solution. Together with the creation of top-level requirements specifications and requirements specifications, system designers also define test plans. These are used later during systems integration and test to be able to determine that the product, as delivered by system suppliers, satisfies the needs of all system stakeholders. As result of the development process, product specifications, also called system supplier specifications (SSS), are created [96]. These specifications are delivered back to the respective system designers, i.e. requirement engineers, for approval and implementation acceptance. In summary, systems suppliers define “how” and “whereby” they plan to solve the design task as postulated by the designers. The SSS also constitutes a binding contract between system supplier and designer [96], [233] and [203].

Figure 2.4 depicts the three types of specifications that have been described before together with the actors that create them. Four general requirements engineering steps need to be passed on the left-hand side and the bottom of the V-Model:

- Requirements Elicitation
- Requirements Analysis
- Requirements Specification
- Specification Validation

As depicted by Figure 2.3 and Figure 2.4, specification validation is a crucial part of the design and development process [47] and [45]. Different definitions exist for the

process of validation, depending of the field of engineering. In the context of this work, the process of validation encompasses all activities to ensure that the SuD, as described by requirements or product specifications, satisfies user specific needs during systems design [243] and [231]. Further, validation processes secure that all necessary use cases can be accomplished with the intended design (i.e. functional requirements), that it satisfies specific non-functional requirements (e.g. performance targets) and constraints [244] and [169]. Thus providing an answer to the question: “Am I developing the right product?” [141]. Validation is performed on the left-hand side of the V-Model and during final transfer of the product to the customer [354] and [169]. According to Easterbrook [107], the process of validation is based on subjective qualifications since it “[...] involves making subjective assessments of how well the (proposed) system addresses a real-world need.” [107]. After all specifications have been validated, usually beginning with the implementation of the SuD, product verification activities are performed against product specifications.

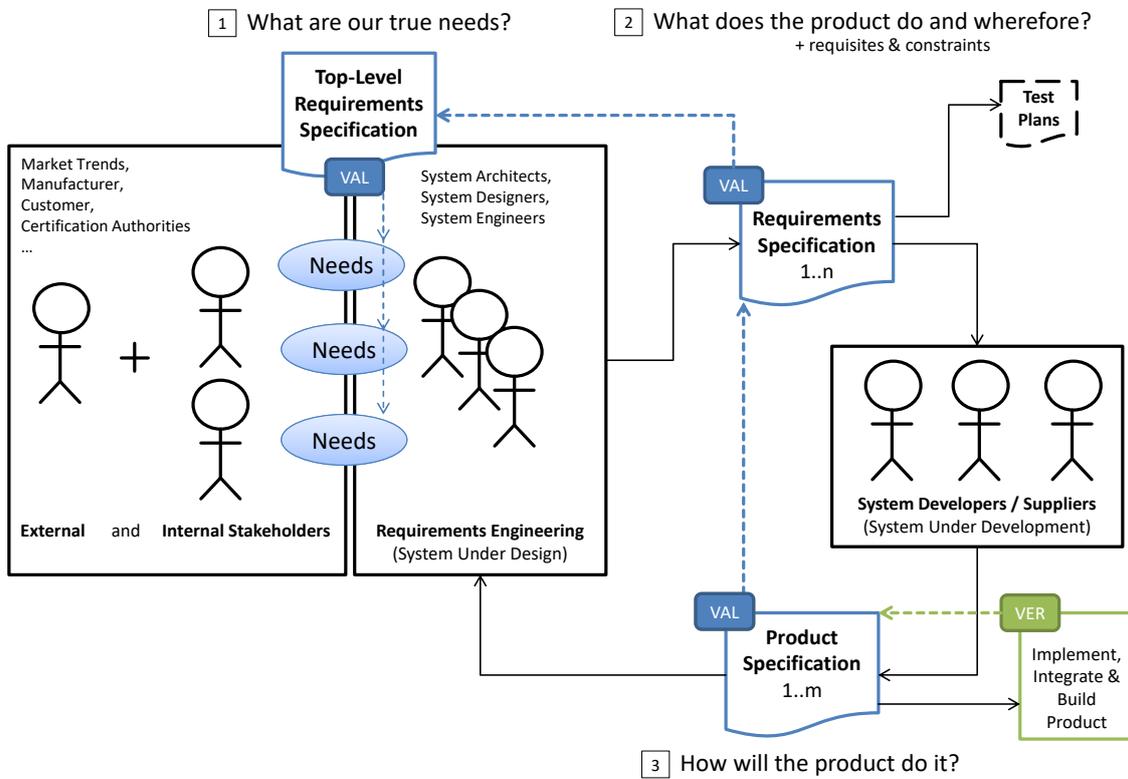


Figure 2.4 – Different actors contribute to the process of requirements engineering and development and form different types of specifications. Specifications need to be validated (VAL) while implementations have to be verified (VER)

Verification on the other side encompasses all activities to ensure that a system has been developed and implemented right [243] and [231]. In other words, verification proves that the results of development, implementation and integration processes correlate with existing product specifications, thus showing that all functional and non-functional requirements have been implemented correctly [169]. Hence the question to be asked by verification actors is: “Am I building the product right?” [141]. Verification activities are performed beginning at the bottom and on the right-hand side of the V-Model process [354] and [169]. Unlike validation, verification is an objective process that requires no subjective judgments in order to determine the

quality of a SuD, provided that the various specification documents are expressed precisely enough [107].

In summary, the most important difference between validation and verification is, that verification in contrast to validation can only help to determine whether the SuD, its subsystems and components (hard- and software) are of good quality and have been implemented correctly. Thus, verification will not ensure that the product will be useful for the intended user [107].

Figure 2.4 shows that different types of specifications have to be validated against different formal or informal descriptions. Top-level requirements specifications represent the initial part of the design process and have therefore to be validated against the true needs of the different stakeholders of the product that is to be designed. Requirements specifications for systems, sub-systems and components are validated against respective top-level requirements specifications to ensure a consistent and complete transition from a more abstract overall system level down to component level. Product specifications developed by system suppliers are validated against corresponding requirements specifications to ensure that the chosen development approach reflects the needs of system designers and that all given requirements are solved completely and correctly. Verification starts with implementation activities performed by the systems supplier.

2.2 Problem Analysis - Product Uncertainty and Risk of Development

“Don’t worry about that specification paperwork. We’d better hurry up and start coding, because we’re going to have a whole lot of debugging to do.”

— Barry W. Boehm on the most prevalent position towards the validation of design specifications during software development in 1984 [47]

Already during concept design for complex system-of-systems, e.g. aircraft, the number of single requirements to meet top-level specifications is large. To speed up development and meet timing objectives, the overall system design is partitioned into a large number of parallel subsystem developments that are independently developed by different departments, divisions and companies involving hundreds of system engineers [316]. Thus, emerging from the overall V-Model development process at aircraft level, several parallel V-Model processes are formed for each subsystem.

Throughout the systems design process for aircraft, the creation and validation of written specifications is typically done using office applications and flow-charts according to the *Department of Defense Architecture Framework* (DoDAF) [241] in conjunction with requirements specification storage and management methods applying tools such as *IBM Rational DOORS* [164] and [297]. Written specifications can contain hundreds or even thousands of single requirement descriptions in natural language that need to be comprehended and validated for unambiguousness, correctness, completeness and consistency [45] by independent validation engineers or designated members of certification authorities [169], currently applying different methods including [263], [264] [318] and [297]:

- Inspection through design reviews by different groups or individuals
- Interviews with system designers
- Re-reading of specifications and utilization of checklists
- Traceability analyses within specifications (top-down and bottom-up)
- Comparison and judgment based on system similarity and experience
- Engineering judgment

The development methodology described within chapter 2.1 is often characterized as *slotted development*, derived from the parallel and independent design and development of subsystems that are eventually integrated. This commonly used development approach was significantly influenced by the British software and systems engineering specialist Michael A. Jackson. Jackson developed the *Problem Frames Approach* (PFA) [171] and [172] derived from his earlier work called *Jackson Systems Development* (JSD) [170]. The PFAs fundamental design philosophy is to focus on the design problem rather than the early solution and to approach requirements analysis and systems design through a process of parallel decomposition and independent development as illustrated by Figure 2.5. Each subsystem is expected to validate and verify its own design to achieve good quality during integration, when the overall system is formed as a whole [171]. In contrast, as depicted by Figure 2.5, slotted development processes for complex systems do not currently achieve the postulated design quality during integration, resulting in the revision of up to 70% of all requirements within specifications [1], [157] and [300].

Since the slotted development process is intended to begin subsystem development as early as possible, the partitioning into subsystems is carried out before uncertainties about individual subsystems and their interactions have been resolved. Therefore, mismatches may exist between (top-level) design assumptions and the solutions proposed by subsystem designers [118] and [72]. Moreover, each of the development groups has a unique knowledge in a particular field. This includes the usage of tailored software tools and methodologies to design specifications and to develop specific hardware or software components. These different tools and methodologies may not be compatible with each other [313]. In the process of development, each of the subsystem design teams will make certain assumptions about how their design will interact with other subsystems. These assumptions often differ from each other and are sometimes not documented. In the case of subsystems that are reactive to their environment, the behavior of the overall system cannot be predicted by the individual designers. Furthermore, insufficient communication between design teams increase the need for additional assumptions since not all required information is passed on [311].

Because of the individual field of expertise for each subsystem, and since it is not possible for one design engineer to understand the complex interactions of the system of systems (SoS) of current aircraft, design groups will consider a unique range of possible or approximate values for parameters during design, validation and verification. This information is written down and exchanged between designers and developers

at different design levels and subsystems and will typically not document all assumptions and ranges for the considered design and thus include uncertainties. The reason is that some of these uncertainties are very specific to the field of knowledge for a subsystem or component and thus cannot be understood or properly evaluated by other designers. It is also not possible for subsystem designers to determine the impact of design decisions and related uncertainties either on the interaction with other subsystems nor on the overall system [313]. In addition, because the development of hard- and software is not coupled (cf. chapter 2.1), software is designed and developed without detailed knowledge of timing and resource dependencies that evolve from the intended system architecture [215].

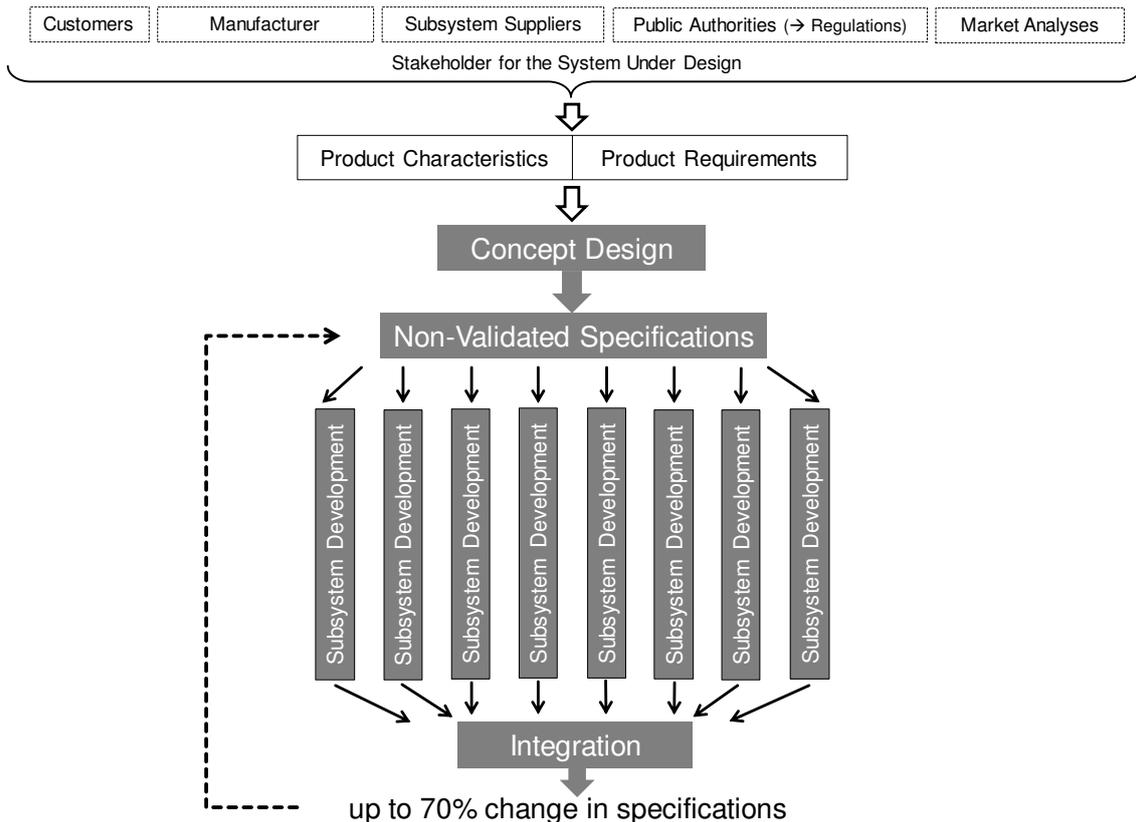


Figure 2.5 – Slotted system development process; from concept design to integration with up to 70% specification changes caused by a high level of design uncertainty due to invalidated specifications (adapted from [125] and based on [1], [157] and [300])

These circumstances provide the prerequisites for an effect referred to as emergence or emergent behavior. Emergent behavior is one of the most distinguishing features of complex systems and is a direct result of the coupling between different parts of a system. It is expressed by the formation of surprising or unexpected behavioral patterns and properties that cannot be predicted from knowledge of smaller and simpler parts of the system taken in isolation [58]. Emergent behavior can also occur in a non-deterministic way during the application of a system, i.e. the system may enter different possible or possibly unknown states for the same set of prerequisites and actions [126].

Eventually, when the developed, tested and locally optimized subsystems are assembled to form the overall system, the system does not work at first. As a remedy, it is tried to fix problems at subsystem or component level with additional testing

procedures. Validation of subsystem design specifications and verification of implementations against overall system requirements cannot guarantee to find design problems since these top-level requirements and specifications may be inconsistent [311]. Thus, if top-level requirements specifications used by different design teams for subsystem development are not validated before commitment in order to be complete and correct, they contain a high degree of uncertainty in the respective requirements. This uncertainty is propagated through the rest of the design and development process. In consequence, vaguely expressed, missing, incomplete or even wrong requirements within specifications cause the need for massive rework in the course of the development process. As a result, iteration cycles for both, design and development are multiplied [313] and [96].

Since specifications, design models and implementations passed on for system integration do not include the described design uncertainties, the overall uncertainty of the integrated system is not determined. Moreover, since no executable overall model exists, coverage analyses cannot be executed for the combination of overall behavior, non-functional requirements and constraints. Therefore, the integrated system that is described by textual product specifications is neither validated nor optimized [311], [313], [316], [317] and [215]. With regard to the task of determining an optimal overall system, i.e. aircraft, it might sometimes even be necessary to fulfill design requirements for subsystems less optimal or Pareto-optimal in order to achieve a global optimum [313] and [30]. A fact that is best summarized by Aristotle's famous conclusion "*The whole is greater than the mere sum of its parts.*"

In a 2013 article [350], Patrick Thibodeau interviewed Jim Johnson, chairman of *The Standish Group*. According to Thibodeau [350], there is currently a very low success rate for the development of large, multi-million US dollar information technology (IT) projects. The Standish Group analyzed 3.555 development projects with labor costs of \$ 10 million and above between 2003 and 2012. According to the article [350] and an analysis published in 2014 [152], the likelihood that the used design methods would lead to a successful system development without critical problems is less or equal 6.4%. Of the remaining projects, 52% were over budget, behind schedule or did not meet customer expectations while the rest were complete failures [350] and [152]. Other studies related to complex systems development that were carried out by analysts of the Standish Group [151], Gartner [221], the National Institute of Standards & Technology (NIST) [300] or the European Software Process Improvement Training Initiative (ESPITI) [1] show, that the probability of critical problems is highest during early design stages and become low at the end of development. Thus, while up to 70% of all design errors are introduced during the first stages of systems design and become part of written specifications, most of these errors are found late during integration and test phases [300]. For the development of airborne systems, John Strasburger from the *Federal Aviation Administration* (FAA) states [343], that most problems that are reported to be software problems or anomalies are due to incomplete or incorrect requirements and not implementation errors. In conclusion, uncertainties introduced during requirements elicitation, analysis and specification phases are the main reason for failed development projects [157].

Design and specification errors that are detected late cause an increase for the required amount of overall development time and development costs [225]. Moreover, errors originating within specifications even pass through to operation phase with

an average of 20.5%, being the worst- case scenario for every customer [46], [146], [117] and [197]. A scenario that is not only hard to measure in terms of costs alone but also by loss of reputation or even the cancellation of a project. In 2013, the *Air Accidents Investigation Branch* (AAIB) investigated emergency ditchings of two Eurocopter EC225 helicopters in the North Sea [55]. They found, that the reason for the loss of the two helicopters were incorrect specifications that were supplied to a company that manufactures pressure switches for the EC225. These specification flaws led to erroneous failure messages that resulted in the ditchings [260] and [55]. As a result of the accidents, much of the global EC225 fleet was grounded by public authorities. The company *Bond Offshore Helicopters* stated [259], that this incident caused costs of £4.17 million (6.74 million US dollar) in only three months.

Already in 1997, the Standish Group declared that in terms of complex software development projects, an estimated amount of \$81 billion U.S. dollars was spent on cancelled projects in 1995, growing to \$100 billion dollars in 1996 [151]. Figure 2.6 depicts the described overall V-Model systems development process including the frequency in percent of where errors are likely to be introduced and found based on the works of Lewis, Feiler et al. in references [117], [116], [115], [120] and [197]. Estimates of nominal costs for correcting design errors are given, whereas the character “x” represents a normalized unit of costs and can be expressed in terms of person-hours, dollars and so forth.

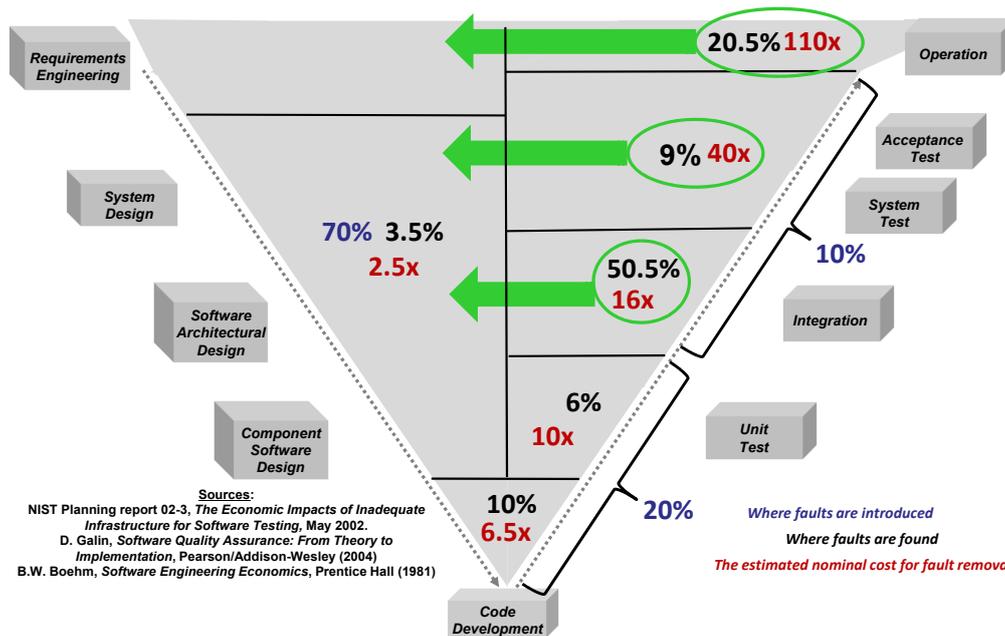


Figure 2.6 – V-Model development process including the percentages of where design errors are likely to be introduced and found together with preliminary estimates of relative costs; green arrows indicate the necessity of shifting the discovery of design errors to earlier development phases (Source: Feiler 2010 [116] and [115], based on [300], [46] and [146])

Other studies, e.g. by Redman et al. [286] or the International Council on Systems Engineering (INCOSE) [250], predict an even higher growth potential for the cost required to remedy design errors during late development stages. According to the INCOSE [250], the cost to extract defects within system designs goes up to a factor of 100 during development and reaches a factor between 500 and 1000 for late development stages through to system operation.

To be able to estimate and quantify the impact of design uncertainty of early design stages on the certification process of aircraft, the following two examples shall be used. Since certification is a critical part for the overall success of an aircraft, significant amounts of systems specifications and certification documentations are produced. For example, the amount of documentation that needs to be produced and verified for certification for one particular type of electronic control unit (ECU) within the aircraft cabin domain comprised 24 different documents with a total of 1380 DIN A4 pages in 2008 [183]. In another example, changes in the development efforts of a major cabin system for two large aircraft developed by the same manufacturer were examined. The efforts for creating the necessary documentation for aircraft certification has increased from 478 DIN A4 pages with a development effort of 923 hours in total in 1997 to about 5217 DIN A4 pages with a development effort of at least 5568 hours in total in 2011 [183]. Uncertainties within product design that are treated late within the development process will therefore impact a potential of thousands of pages necessary for certification within one subsystem alone. An equal amount of necessary changes can be expected for specification and implementation.

Research in the field of system design shows, that most causes for critical design flaws stem from poor design specifications (top-level specifications and related requirements specifications) [44], [313] and [316]. The low quality of design specifications was traced back to the inability to validate specifications during early design stages. The reason behind this is the exclusive usage of textual specifications during development [157], [311], [313], [316], [317], [215] and [96]. Salzwedel et al. [313], [316], [317] and [215] examined the occurrence of product uncertainties and development risk during complex systems development. Figure 2.7 depicts the change of product uncertainty within the currently used slotted development process together with the progression of development costs over time. In the course of development, product uncertainty is usually expected to decrease (red line). But the late detection of specification errors during integration that results from coupling between subsystems, software and hardware leads to an increase of product uncertainty back to previous levels of conceptual design (dotted arrows) [317] and [30].

By comparing Figure 2.6 and Figure 2.7, we observe that the occurrence of critical design errors, product uncertainty and development cost growth are highly correlated. In 2007, Salzwedel [313] analyzed the occurrence of critical design errors and product uncertainty for complex system developments and stated: “*We may conclude that critical design errors are primarily due to information uncertainty about a product and design processes that do not consider this uncertainty in the right way and do not validate and verify the system considering this uncertainty.*” [313].

According to Salzwedel et al. [313] and [316], product uncertainty U_P can be expressed by:

$$U_P(t) = U_D(t) + U_L(t) - L_S(t) \quad (2.1)$$

Where:

- U_D is the learning curve for the product during development.

- U_L is the approximation for additionally introduced product uncertainty during the lifetime of a product due to lost design information, personnel discontinuity and people that leave development teams, changes in regulations or the usage of the system for applications that were not considered during design.
- L_S is the learning effect due to customer testing and support after the product has been delivered.

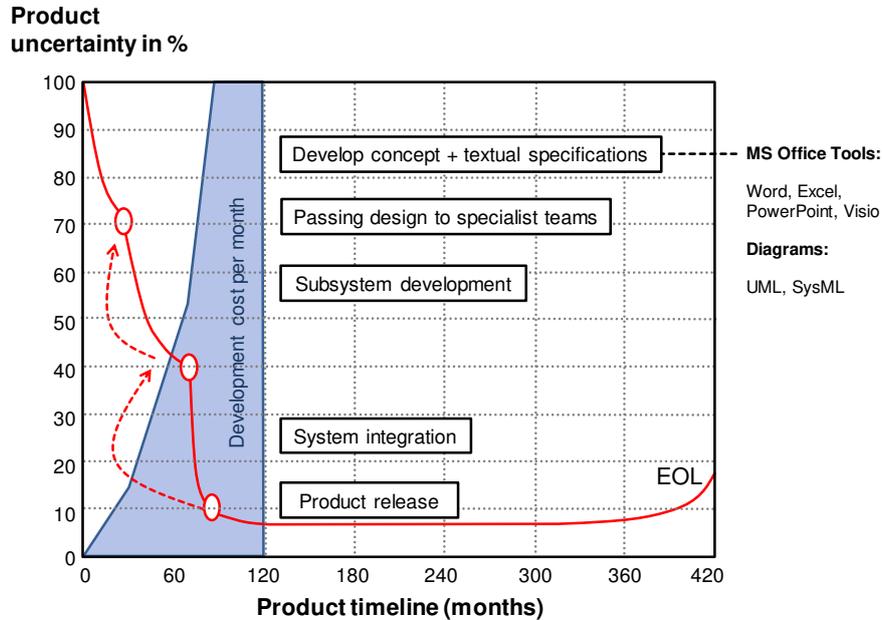


Figure 2.7 – Product design uncertainty and development costs progress during aircraft development with regression during integration leading to a growth of product uncertainty and expenditures (adapted from Salzwedel et al. 2008, 2009, 2010 [316], [317] and [215])

According to Salzwedel et al. [317], the financial risk of product development can be considered together with product uncertainty. The development risk R can be expressed by [317]:

$$R = \int U_P(t) * C_M(t) dt \quad (2.2)$$

Where $C_M(t)$ is the maximum development cost per month. Figure 2.8 shows a Lorenz curve for committed costs as part of an ABC analysis [154] for the life cycle costs (LCC) of a product accrued over time together with coefficient growth for expenditures to extract defects within the design of a product. It is based on a statistical analysis of the *U.S. Department of Defense* (DoD) that was reported by the *Defense Acquisition University* in 1993 and was republished by the International Council on Systems Engineering (INCOSE) in 2010 [250]. Bars along the time line represent the accrued percentages of actual LCC for the respective product stages. The curve for committed costs represents percentages for the amount of LCC that is determined by project decisions during each stage of the product development cycle [250]. By comparing Figure 2.8 with Figures 2.6 and 2.7 we observe that the cumulative percentage of life cycle costs and committed costs for product development are also highly correlated with the observed occurrence of critical design errors and

the amount of product uncertainty. In addition, we observe that while an average of 8% of the actual cost has been accrued during concept stage, 70% of the total LCC have already been determined [250].

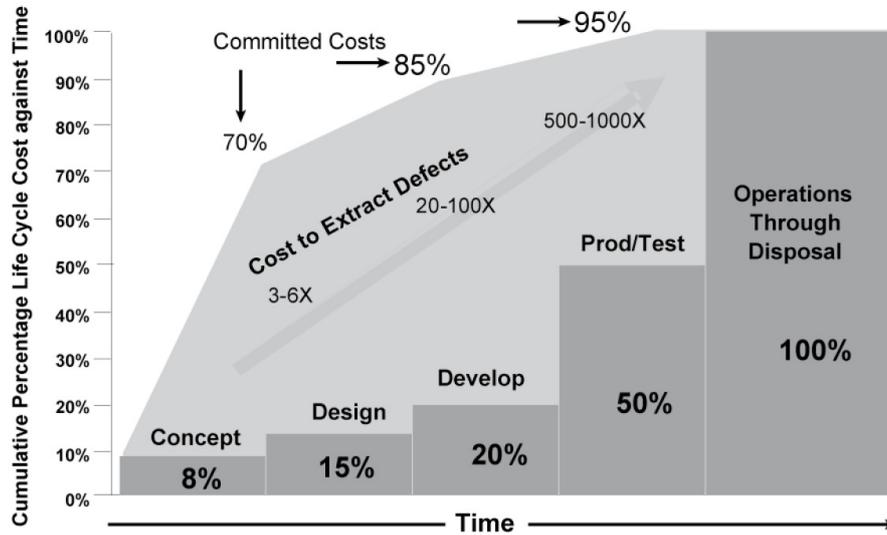


Figure 2.8 – Committed life-cycle costs during development (Source: INCOSE 2010 [250])

In conclusion, development risk is lowest at the beginning of a product's life cycle because accrued LCC and cost to extract design flaws are still low and culminates during product operation. Moreover, the majority of committed costs are determined during early life cycle stages. That implies, that the later specifications are changed the higher the financial risk and therefore the higher the risk that a project will fail [359]. Because the current development process solves critical design problems when costs for changes are at a maximum, it maximizes the overall development risk and can therefore be considered a high-risk development process [313] and [215].

The following issues with the currently applied systems development process for complex systems, i.e. aircraft, can be summarized:

- The communication of design intent and desired behavior is difficult for all stakeholders of a product under development [72]
- The amount of distinct information within requirements and specifications for aircraft and aircraft systems is huge and hard to comprehend through reading and reviewing during validation
- Mismatches exist between (top-level) design assumptions at the beginning of a project (what the customer wanted) and the actual design and requirements descriptions within specifications (what the engineer designed) [118]
- Too much focus is being placed on producing textual specifications and documentation [72]
- Coupling between hardware, software and subsystems is not covered by applicable certification guidelines for civil aircraft such as the DO-254 and DO-178B

- As a result, software is being developed without detailed knowledge about available resources and constraints imposed by the intended system architecture [215]
- Subsystem partitioning is performed before top-level requirements specifications have been validated at overall system level to resolve uncertainties about individual subsystems and their interactions [313] and [316]
- Subsystem developers cannot determine the impact of their decisions on other subsystems and the coupled overall system [313] and [316]
- Early system development phases account for most critical design decisions and introduce the highest amount of product uncertainty (>70%) during development [151], [221], [300], [1] and [359]
- Starting from early design stages, a high amount of product uncertainty is propagated from specification development to implementation and integration phases or even beyond entry into service, because the majority of all design errors are identified too late [359]. Thus, overall design uncertainty is not decreased during the first stages of systems development. As a result, the amount of iteration cycles needed to complete system development is increasing [300], [1], [157] and [322]. As a consequence, many systems have to be redesigned at high cost.
- In the case of dynamic coupling between subsystems, solutions cannot be found during integration. As a result, the overall development needs to go back one or several levels in the development process and do a redesign.
- The probability that the currently used development approach will succeed without critical errors is less than 6.4% [350] and [152]
- Studies indicate, that most causes for critical design flaws for complex systems stem from poor design specifications [311], [313], [316] and [317]
- The low quality of specifications is a result of the inability to validate specifications early in the development process because specifications are on paper only [311], [313], [316], [317] and [215]
- The currently used development process focuses on bottom-up driven design solutions that cannot be used to optimize products for top-level requirements
- Since no overall model exists for the system under design, the overall system can neither be validated nor optimized [313], [316] and [317]
- Current development approaches maximize development risk with regard to the related increases in expenditures and time required for the successful completion of product development [313] and [215]

Since most design flaws for complex systems development arise out of the inability to validate specifications early during development and thus strongly contribute to the propagation of invalidated designs to development, implementation and integration, this issue is also referred to as *Specification-Validation Gap* [126] and [188]. To be

able to solve current problems of complex systems development described within this chapter, e.g. for highly configurable avionics systems, the primary goal of any development process needs to be the reduction of the overall risk of development.

2.3 Existing Solution Approaches

A diversity of approaches may be applied at different stages of the life cycle within the field of systems engineering in order to contribute to the successful development of complex systems such as aircraft. This chapter starts with a description of how to design the “right system” with regard to all stakeholders. This includes an analysis of the difference between creating specifications with improved quality and producing the right (sub-) systems. Moreover, this chapter provides a detailed analysis of existing solution approaches and related work.

2.3.1 Developing the Right System

Companies, system developers and a wide variety of other individuals involved in the development of aircraft or any other product pursue an ultimate goal: to provide customers and other stakeholders with a solution that serves their needs best. In other words, the ability of any system provider to supply stakeholders with the best solution for the intended purpose and application (mission) of a product determines the success or failure of the product [215], [127], [126] and [128]. Figure 2.9 illustrates the general goal of system development together with requirements that enable this goal, i.e. the design and realization of the “*right system*”. To find an optimal solution for any system under design (SuD), it is essential to truly understand, extract, specify and implement stakeholder needs in their entirety from a developer’s point of view.

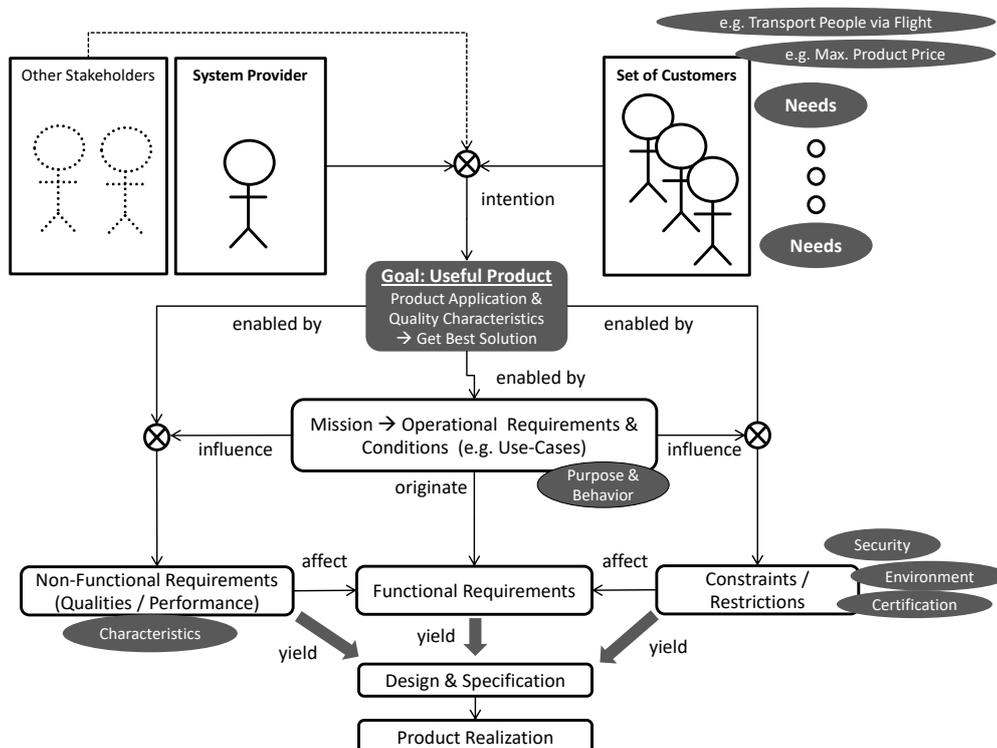


Figure 2.9 – Developing the “right system” in the course of system development

In the first place, a product needs to support a set of customer-related missions and quality targets that result from operational requirements, system context and a set of use cases. Based on the overall mission or intended application of a product, sets of functional requirements as well as non-functional requirements are derived, including system costs, weight, power consumption, durability or availability to name but a few [108], [301] and [370]. While functional requirements determine a certain behavior for the SuD, non-functional requirements determine certain qualities and performances associated with overall system behavior. Both types of requirement categories are complemented by system constraints. System constraints may arise from internal or external stakeholders such as customers (e.g. system architecture configurations), certification authorities (e.g. safety regulations) or the operational environment of the SuD (e.g. physical conditions). A specification is developed that unites all three sets of requirements in order to determine a feasible solution for the SuD. After successful specification development, the SuD is implemented, produced and put into service.

2.3.2 Designing Systems Correctly

As early as the 18th century during the *Age of Enlightenment*, Immanuel Kant created one of his most influential works called *Critique of Pure Reason* (German: *Kritik der reinen Vernunft*) [178]. Within his work, Kant looked closely at existing concepts of philosophy and metaphysics, trying to unite reason with experience [178]. Knowledge, as perceived by the movement of critical rationalism (René Descartes, Gottfried Wilhelm Leibniz), arises from purely rational cognition, i.e. a priori knowledge inherent to all things that can be accessed by humans through analytical analyses [61] and [179]. In contrast, as perceived by the movement of empiricism (John Locke, Francis Bacon), knowledge is gained a posteriori, caused causally by sense experience [60] and [179]. Kant stated [178] that only the unity of both sources of knowledge, analytical reason and experience, can lead to a holistic definition of knowledge and cognition within all sciences.

If *Kant's* principle is interpreted and used analogously for the field of systems design, we may observe the following. In order to be able design correctly, the “right system” needs to be provided for a set of stakeholders at the end of systems development as described within the previous chapter. Therefore, in the first place, a system under design needs to be considered holistically, combining mission, functional and non-functional aspects as well as constraints to gain access to the necessary knowledge needed for comprehensive design, validation and development. Consequently, specifications also need to unite all necessary information on functional and non-functional requirements as well as constraints in order to define a useful product (cf. chapter 2.3.1). To ensure that specifications represent the system to be developed, it is equally important for the process of validation to be based on the same principles of holistic knowledge unification.

In the second place, as most system developers, e.g. aircraft manufacturers, have developed similar systems before, existing experience is to be used as input for both, design and validation of novel system architectures in terms of reusability. Analogously, specifications or parts of specifications can be enriched, substituted and validated with valid knowledge from previous developments. This can be useful if knowledge about a particular aspect for the system under design is not yet available,

unclear or fuzzy. Another important issue is the investigation of coupling of system functionality (software) and architecture (hardware) for the purpose of specification validation. In that case, one may analyze if the combination of required functionality with already available architecture components provides a feasible solution or leads to emergent behavior. Experiences from extrapolated or measured data, e.g. for data transmission, worst case execution times, power consumption and so forth can be used to update specifications with real data. In summary, it may be concluded that only by combining a priori and a posteriori knowledge developers will be able to design correctly in terms of developing validated specifications for the right system.

According to Schorch [323], two categories can be distinguished as causes for erroneous system designs during development, occurring separately or in combination:

1. Incorrect or insufficient specifications
2. Incorrect implementation of specifications

Specifications are incorrect or insufficient if, despite flawless implementation, integration and test, the system does not meet customer expectations [323]. Incorrect specifications can therefore either be incomplete, i.e. critical user needs have not been included, or wrong, i.e. requirements included within specifications or emergent behavior caused by coupling effects are not intended by the customer. The incorrect implementation or interpretation of specifications on the other hand may occur any time during system development, integration or test. A system is to be considered designed correctly if specifications are complete and correct, thus representing the “right system”, and the system under development is an exact implementation of all specifications [323].

Hence, the realization and production of appropriate (sub-) systems can only succeed, if specifications are correct before commitment. In addition to *Schorcht's* definition for the correctness of specifications, Swatman states [188] and [346], that system developers need to ensure that:

1. Specifications reflect a system which meets the real needs of the users in an appropriate way
2. Specifications are sufficiently precise to form the basis for development and construction of the system

What can be done to improve the quality of system specifications, especially for aircraft, in order to reduce development risk and associated problems as stated within chapter 2.2? To be able to improve specification quality, product, design and subsystem interaction uncertainties need to be reduced as early as possible during systems development, including the early coupling of hardware and software design. This is essential to be able to solve design problems for aircraft and airborne systems during design, when costs are low and not during development, integration and test when costs boom. In other words, it is crucial, that specifications are suitable and complete from the users' point of view, including as little design errors as possible (cf. summary at the end of chapter 2.2). This can only be achieved, if

specifications are validated early within the development process in the presence of bounded and statistical uncertainties, thus closing the *Specification-Validation Gap* [118], [1], [157], [197], [316] and [323]. Additionally, in order for specifications to be sufficiently precise as basis for validation, implementation, integration and test [188] and [346], they need to be unambiguous and verifiable [263] and [264].

One way to form the basis for precise and unambiguous content containing knowledge is the application of formal methods and notations. However, the application of formal methods does not necessarily require the use of a specific logic or mathematics. In systems engineering and computer science however, the phrase formal method is more specifically referring to the use of a formal notation to represent models of systems during development [208]. These models have the major benefit to provide an almost complete coverage of the real-world system within the borders of the model [69]. Another advantage of using formal notations with definite syntax and semantics is that they are machine-readable and testable [305]. Formal notations can have different manifestations, including graphical, textual, or mathematical notations. Good examples that are used mainly for the formal specification of functional requirements for software and computer-based systems are the *Unified Modeling Language* (UML) [174], statecharts e.g. *Finite State Machines* (FSM) [147] or the *Z* notation [338].

Since current specifications for aircraft are produced textually on paper in terms of natural language, they cannot be considered formalized and can thus neither be regarded machine-readable nor can they be validated with currently available computer-based methods. Thus, it is proposed, that the creation and validation of specifications within the currently used design process needs to be formalized by applying formal methods, e.g. computer-based methods that have been proven to improve the quality of design and requirement descriptions within specifications in order to be unambiguous and verifiable [155], [208], [63], [254], [144].

However, formal methods alone are not suitable to cover all fields of systems design, e.g. for user-interface design. They can also not be used standalone in a way to replace the current systems engineering process. Instead, they need to be integrated within existing development models as tool for the description of requirements within specifications and for the process of validation [51]. Another disadvantage of using purely formal methods and notations for specification and validation is that they may not effectively handle the development of large and complex systems [208] and [262].

Various methodologies exist to validate system designs determined by specifications. For less complex systems or software, so called agile development techniques [214] can be applied that use quick iteration cycles of implementation and test together with regular meetings to simultaneously validate requirement specifications and build the system under design [84]. In the case of hardware development, prototypical parts are built and tested for validity as early as possible during design. Rapid Prototyping [288] and Digital Prototyping [150] methodologies are prototyping representatives for validation and verification that have been proven to be successful [288].

Another possibility is the usage of models which are an abstraction of a design problem for a particular purpose [339] and [340]. These models can be validated and tested through simulation before the system under design is implemented [340] and [271]. Models include logic models, physical models and analytical or computational

simulation models [242] and [294]. The latter are often created with the aid of computers and include differential equations, algebras or block-oriented simulation models [314], e.g. Finite State Machines [121], Discrete-Event Simulations (DES) [27], Stochastic Discrete Event Systems (SDES) [386] or Discrete Event System Specifications (DEVS) [385] to name but a few. Holt et al. [161] provided a theoretical and more detailed description on how to perform the process of requirements engineering on a model-centric basis, called “*approach to context-based requirements engineering*” (ACRE), in their book “*Model-Based Requirements Engineering*” [161].

One major advantage of specifying requirements and systems designs based on graphic notations like UML arises from the human nature. Humans already access knowledge by creating abstract models of the perceived reality. Cognitive research findings also show that humans tend to easier understand information and solve problems when using imagery [184] and [148]. According to Pohl and Rupp [268], these findings can also be applied to requirements models [268]. Another advantage of using a graphical notation over natural language texts is the provision of a well-defined level of abstraction supported by a set of predefined language elements for specific system design tasks [268]. Moreover, languages like UML or SysML provide common extension mechanisms to include new language elements [326]. Cultural complexity with language as well as cultural barriers is also a major challenge in aircraft development [12]. Here, the use of imagery may contribute to finding a common ground of understanding during conception and systems design.

The *International Council on Systems Engineering* (INCOSE) founded in 1990 is a non-profit organization dedicated to the advancement of systems engineering. In recent years, the INCOSE analyzed and developed guidelines concerning specification creation and validation methodologies for complex systems development. The council identified the usage of computer-based design tools for forming formal simulation models as the key technology for future improvement for validation during systems development within their roadmap from 2010 to 2035 [144], [112] and [35].

A design and validation method that is primarily based on computer models and simulations is often referred to as *Model-based Engineering* (MBE) or *Model-based Systems Engineering* (MBSE) [112] and [250]. As part of ongoing systems engineering research, INCOSE established a dedicated model-based systems engineering initiative in 2007. An overview of different MBSE methods and practices analyzed by the INCOSE is provided in reference [112]. MBSE may also be referred to as Model-driven Engineering (MDE) because of the pivotal role of models within the systems engineering process. In the context of this work, the definition for the term *Model-based Systems Engineering* is taken from the INCOSE Systems Engineering Vision 2020 [246]:

Definition 1 “*Model-based systems engineering (MBSE) is the formalized application of modeling to support system requirements, design, analysis, verification and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases.*” [246]

Another term used for the application of executable computer models for systems development is *Simulation-based Systems Engineering* (SBSE). The fundamentals of SBSE have been covered in literature, e.g. by Clymer [81]. Clymer states [81], that

the application of formal models and model execution, i.e. simulation, is the key factor to understand complex system behavior and structure. Understanding of the whole system, especially in terms of emergent behavior, cannot be gained from the knowledge that exists about each of the independent system components. Clymer also states [81], that simulation is required to design, evaluate and optimize complex systems.

A good example that supports Clymer's statement is the potential number of interactions that exists in a complex system that is composed of a substantial number of elements. According to Altfeld [12], the number of potential interactions N among a number of n modules approximately doubles with every new module added and can be described by [12]: $N = \sum_{i=2}^n \frac{n!}{(n-i)!i!}$ If analyzed on paper only, the effects on the overall system design are hard to be determined by individual system engineers.

Within their final report in 2011, the *model based engineering subcommittee* of the systems engineering division of the *National Defense Industrial Association* (NDIA) [227] concludes, that the field of model-based engineering is one of the four most important research areas for the next 20 years that needs to be improved [35] and [34]. NDIA also states [34], that it fully supports the MBSE assessment made by the INCOSE. For the process of acquisition and the awarding of contracts to possible system providers for complex national defense systems, the *United States department of defense* (DoD) developed guidelines that made the application of modeling and simulation methods mandatory for system development, validation and test [347].

An indication of the possible benefits of using computer models and simulations as part of the development process for aircraft can be found within other areas of design. According to Altfeld [12], the conceptual design of aircraft has greatly benefited from *Computer Aided Engineering* (CAE), i.e. the usage of digital design, simulation and analysis tools [12]. Mathematical simulations are performed for various engineering disciplines, including aerodynamics with computational fluid dynamics (CFD) and stress analysis by means of finite element analysis (FEA) [12]. Another major advancement is the application of three-dimensional design and animation tools instead of technical drawings on paper, called computer aided design (CAD) [225] and [12].

In the field of CAD, there has been a major change from static graphical models towards dynamic real-time capable simulations and virtual reality (VR). During a personal communication in 2013, the former senior aircraft interior designer at Airbus, Thomas M. Bock, described the changes in aircraft interior design and validation over the past 50 years [42]. Bock emphasized that first of all, the complex structures of aircraft cabin interior itself, consisting of all kinds of shapes, dimensions, colors and textures are hard to specify and validate by means of textual descriptions and handmade drawings only. This is especially true if knowledge of different pieces of drawn design specifications need to be assessed at system level in collaboration with all kinds of different stakeholders from different technical backgrounds. Moreover, the operating procedures that are related to the usage of different cabin equipment as well as the behavior of cabin components during use or general aspects of usability can only be validated by using executable design specifications. Today, real-time capable VR computer models are used in order to specify and validate cabin designs dynamically [42].

One approach from the range of available MBSE methodologies is the usage of executable specifications (ES) [311] and [124]. In the context of this work, an executable specification is defined as follows:

Definition 2 *An **executable specification** represents a system specification for a system under design, including functional and non-functional design aspects, that is created in the form of a formal, computer-aided model that can be executed and validated by means of simulation.*

Executable specifications that combine mission and operational as well as maintenance and usability models including *human-machine interfaces* (HMI) [126] are also referred to as customer level virtual prototypes or conceptual prototypes [314]. In the context of this work, a virtual prototype defined as follows:

Definition 3 *A **virtual prototype** is an executable specification that is combined with human machine interface concept models. Virtual prototypes can be executed in a simulator and are operated interactively by humans for validation, test and training purposes.*

The application of ES has been successfully demonstrated for the development of subsystems for automotive [281] and airborne systems including the Boeing B777, the Airbus A380 [124], the Lockheed Martin F-22, or the Boeing P-8A of the Multi Mission Maritime Aircraft (MMMA) program [314]. Especially in terms of specification validation, the usage of executable specifications at overall system level has been shown to be most successful [323], [317] and [198]. The characteristics and usage of ES as well as their application in the context of this work and in relation to related work will be elaborated more extensively in chapter 4.

Since the development process of complex systems is phase-oriented and iterative [322], e.g. based on the V-model, different design and development steps have to be performed repeatedly. In order to find the best possible solution for a complex system under development, e.g. for aircraft, a large number of different designs need to be integrated and validated as part of an iterative process [313]. Already in 1979, Boehm analyzed and evaluated different techniques for validation and verification of small and large specifications. In the case of large and complex systems he concluded, that in order to achieve good specification quality during development, i.e. specifications with minimal product uncertainty, validation needs to be automated [45] and [47]. Moreover, finding an optimal solution between sets of different possible designs is an impossible task if done manually. Therefore, it is also necessary to automate the design and validation process [313] and [215]. As a result, a simulation technology is required that is capable to support the creation of directed process graphs of modules that can be independent simulations as part of design and validation methodologies [313] and [316].

2.3.3 Model-Based Systems Engineering: From Lilienthal Until Today

A discovery made in 1900 near the Greek island of Antikythera located in the Mediterranean brought to light a most complex apparatus from the ancient world

now known as the *Antikythera mechanism*. The mechanism dates back to the first century before Christ. Enduring research revealed that the mechanism was a complex and executable model for the prediction of the movements of sun and moon as observed from earth [142]. It may be considered one of the world's first computers and represents a milestone on the way to modern technology for modeling and simulation in terms of system animation and validation. Moreover, it demonstrates that mankind has used executable models to access complex knowledge for a long time [213].

What has been done in the field of model-based systems engineering to solve the challenges stated within the previous chapters? Taking a look back into the history of aviation, model-based design methodologies were key technologies that made aircraft development possible in the first place. During the 19th century, developments for aircraft stagnated and failed until the Lilienthal siblings closely observed the flight of birds and used detailed test data from airfoils attached to a rotating arm to generate relative wind velocity for wings to develop validated models of aerodynamics [199]. Eventually they were able to fly with technology “*heavier than air*” [199] as depicted by Figure 2.10. Following generations of aircraft developers such as the Wright brothers extended and validated these models further through wind tunnel experiments [377] and [378]. In the following years engineers computed aircraft aerodynamics by hand until in 1936 Konrad Zuse developed computers, starting with his first model *Z1*, that were able to perform these calculations automatically, validating the coupling between aerodynamics and structures [5], [389] and [314].

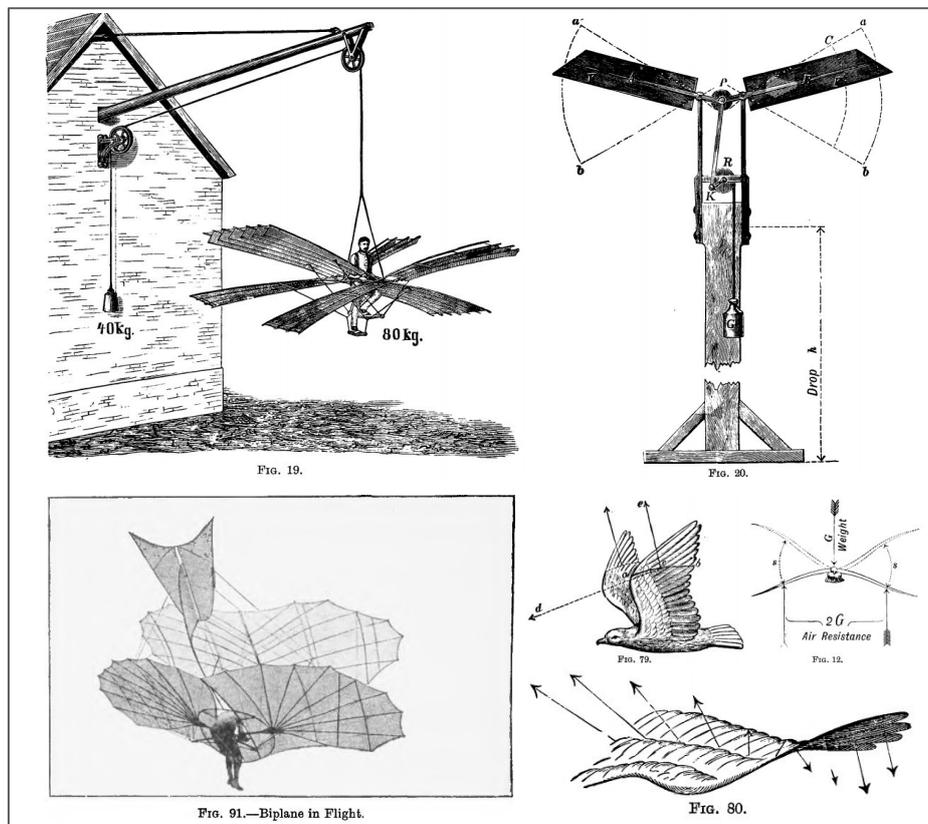


Figure 2.10 – Model-based analysis and development of aircraft during the 19th century, performed by the Lilienthal siblings, based on the observation of birds (adapted from Lilienthal 1889 [199] and [200])

Aircraft development progressed until the 1960s and 1970s with the introduction of stability augmentation systems [311]. These were further extended as aircraft crossed the sound barrier in the 1980s [315]. Every prototype that was tested at this time exhibited aero-servo elastic problems. Analyses showed that flawed specifications were the main cause. These resulted from insufficient communication between designers and a general trust, that verification activities for implementations were sufficient to avoid mistakes in systems design [311]. A prominent aircraft affected a few years later was the *Saab JAS Gripen 39* which crash landed on its first flight in 1989 because of specification design errors within its control systems and insufficient specification verification [342]. During the 1970s and the early 1980s rapid progress in integrated circuit (IC) development doubled complexity every two years as predicted by Moore in 1965 [226]. At that time, ICs were designed on physical level using *Computer Aided Design* (CAD) [29]. Since chip complexity increased by a factor of 100 during the 1980s, engineers were unable to handle the design and validation any more [311]. As a result, chip design abstraction was raised to the logical design level and ICs were designed utilizing hardware description languages and block diagrams [29], e.g. for the development of integrated controllers [94]. Two of the best-known hardware description languages are *VHDL* [24] and *Verilog* [351] which enabled the development of chip design masks. Also during the 1980s, the *Embedded Computer System Analysis and Modeling* (ECSAM) method was developed by Lavi and Kudish for the design of embedded systems as well as for the *Israel Aerospace Industries* (IAI) [191]. ECSAM combines block diagram-oriented modeling and analysis of system structure and behavior. The method also provides capabilities for dynamical analyses of the behavior of systems [190].

In the course of multidisciplinary research during that time, modeling and design methodologies were developed that were able to combine models at functional level from different disciplines. The research was funded by the *Air Force Wright Aeronautical Laboratories* (AFWAL) [311]. *System Control Technology* (SCT), a division of *Science Applications International Corporation* (SAIC) developed common modeling and simulation environments called *Ctrl-C/ModelC* that supported the necessary functional level design and validation methodologies. A similar modeling and simulation environment developed was *MATRIXx* (*Integrated Systems, Inc.*, now part of *National Instruments*). With that, formal mathematical descriptions could be formed for aerodynamics, structures and control in one common executable model to describe, simulate and analyze aircraft dynamics [309], [308], [311] and [215]. To be able to develop these methodologies and tools, the existing control theory needed to be extended. Research and developments of that period also sparked the development of tools that are still in use like *Matlab* or *Octave* [315].

A good example of the application of the developed design and validation flow in conjunction with the developed tools is the AMRAAM system (*Advanced Medium-Range Air-to-Air Missile*). Its multidisciplinary development involved the coupling of different disciplines (flexible aircraft structure during flight including navigation, missile system and missile alignment) and different subsystem suppliers. The overall system was developed through subsystem decomposition and individual development of each subsystem in isolation. This approach did not permit to optimize the overall system with regard to coupling of mission, aircraft and missile subsystem. When aircraft manufacturers integrated the AMRAAM system within aircraft of the F15 and F16 series, flight tests revealed that both alignment times and navigation errors

for the missile system were ten times too large with respect to the intentions of the customer. These problems were only solved, after integrated and coupled tools were used within one multi-disciplinary design flow including mission, flexible aircraft and control model. As a result, a transfer alignment filter was developed that made the missile system not only usable for aircraft for the first time but also one of the most successful systems of its kind [311] and [315]. In the end, the accuracy of the coupled aircraft-missile-system could be improved by more than two orders of magnitude while simultaneously reducing the alignment time again by more than two orders of magnitude [309], [308] and [311]. In 1988, it took two to four hours to analyze one operating point and 100 to 150 hours to analyze and validate the design for all axes for missile flight controls. By using a formal modeling and simulation approach, the analysis and validation of all parameters, operating points and all axes took only one hour in total [69].

Since system complexity increased again during the 1990s by two orders of magnitude, modeling and simulation tools like *COSSAP* developed by *Synopsys* or the *Signal Processing Work-System* (SPW) developed by *Cadence Designs Systems* were used to raise the abstraction level for model-based IC design and validation to the functional level [311] and [29]. Hardware description languages are still considered for avionics systems development, more specifically for the design of hardware components at lower levels of the V-Model. But still, the process of validating specifications within the concurrent development process of hardware, software and systems coupling could not be sufficiently solved for avionics [261]. As early as in 1962, *Boeing Satellite Systems* (BSS) began working towards simulation based validation methodologies for spacecraft today known as *Integrated Development & Validation Process* (IDVP). Facing the growing systems complexity challenge, Boeing realized that hardware and software development based on early validation is most critical for success [332]. A directive was defined to create a prototypical simulation environment to, as Slafer points out [332], “*fly spacecraft on the ground*”. Hardware and software were developed with a mixed validation, verification and test strategy for system designs and performance evaluation using hardware prototypes and computer aided simulation. Boeing thus achieved a percentage of 100% for the success of every first mission on all developed Boeing spacecraft [332].

With the movement towards networked system architectures starting in the early 1990s, model-based functional level design and validation methodologies were reaching their limits. The number of electronic control units (ECUs) exceeded 100 for civil aircraft. Because of the coupling of different components within systems, behavioral level models were needed to validate requirements specifications [314]. To be able to validate system architecture designs with hundreds of ECUs and different data networks, the *United States Department of Defense* (DoD) got a company called *MIL 3 Inc.* (now *OPNET Technologies Inc.*) to develop an object-oriented network simulator called *OPNET* in 1985 [315]. Coined from a military background, the tool began to spread to systems engineering and is still in use today for the development and validation of static system architectures [272]. In the case of networked, dynamic aerospace systems however, *OPNET* and related design methods failed to meet necessary requirements for specification validation, dynamic architectures and performance analyses. AFWAL and Rome Air Development Center (RADC) therefore commissioned the development of a new platform capable of designing and validating dynamic system architectures for aircraft. As a result, *Cadence De-*

signs Systems developed the *Block-Oriented Network Simulator* (BONeS Designer) in 1989. By combining *C* and *LISP* programming language features within a common design and validation platform and validated libraries it was possible to design and validate parallel, distributed and dynamic systems architectures at performance and behavioral level [315] and [327]. By combining *BONeS Designer* with an additional tool by the same software developer, called *SatLab*, it was also possible to design and display satellite systems [192].

One of today's most successful aircraft in terms of reliability and sales figures, the Boeing B777 [109], began its success story in the late 1980s and early 1990s by incorporating eight major airlines into the design process. Up to that point, aircraft were usually designed by manufacturers with minimal customer involvement [366] and [37]. Moreover, the B777 was the first aircraft ever to be designed fully digitally by applying computer aided design methodologies [109], [236] and [369]. The B777 was also the first networked civil aircraft [225] and [224]. Since control information is being passed through this network, subsystems are coupled dynamically. The classical bottom-up development process would therefore fail to provide an optimal design solution at aircraft level. In order to avoid this failure and to be able to cope with dynamic system architectures, combining functional and non-functional system aspects, Boeing's B777 was also one of the first commercial aircraft developments that used model-based design methodologies and tools for specification development and validation during the 1990s [315]. By using design and validation tools like *BONeS Designer*, performance level executable specifications could be created, simulated and validated for aircraft networks and avionics at overall system level [315] and [38]. Boeing used *BONeS Designer* in order to develop a Flight Control System (FCS) with deterministic behavior that also assured safety during system failures [315] and [353]. With this, the development time could be reduced by factor two [314]. Additionally, the number of design rework cycles needed to reach a maturity level high enough for entry into service were below two [12], compared to four to ten cycles required during previous aircraft developments [90].

EASY5, the result of more than ten years of development, was also one of the first computer aided simulation tools for dynamic systems provided with a graphical user interface appropriating block elements to help engineers create, validate and visualize complex specification content [98]. A good example is the development of *PlaneNet*, the first fiber optics network system for civil aircraft [15]. Boeing's computer and executable model aided design and development process was successful. As a result, initially build physical mock-ups of the B777's nose section, used to verify the development, were single realizations. All additionally planned mock-ups were canceled [235]. At the same time, the *National Aeronautics and Space Administration* (NASA) worked in conjunction with Boeing to start to move reliability and safety analyses to early systems design stages within a model-based design and validation approach [230] and [353]. This was essential for the development of novel safety critical system architectures, e.g. for the introduction of the first primary *Fly-by-Wire* flight control system for commercial aircraft starting with the Airbus A320 [25]. However, in contrast to the Boeing B777, the A320 is not a networked aircraft since it uses dedicated communication lines for each subsystem [224].

On the other hand, while model-based hardware and networked systems design and validation progressed, software still played a small role and was thus not part

of the model-based validation process [56]. Furthermore, model-based design and validation techniques were used at subsystem level rather than at overall system level [314]. Electrical flight control systems (EFCS) for example, being systems with the most stringent requirements for safety, were developed and tested with the aid of computer simulation tools. The introduction of control and monitoring principles for hardware units and a high level of redundancy assured that these systems were fault tolerant and safe. Simultaneously, software development was considered part of a peer reviewed design and validation process with written specifications [56]. Thus, software implementations for aeronautics were realized and verified according to DO178B [273] or IEEE Std 1228-1994 [245] but not validated beforehand, e.g. with executable specifications. Brière and Traverse stated [56], that in the case of the EFCS development for the Airbus A320, A330 and A340 families: “*For this “functional” part of the software, validation is not required as covered by the work carried out on the functional specification*” [56].

In terms of software architecture development and verification, the *Defense Advanced Research Projects Agency* (DARPA) began investing in research during the 1990s to cope with the existing need to understand the increasing complexity caused by software and hardware interaction within large scale-systems, i.e. aircraft. As a result, different formal software architecture description languages emerged which supported the quantitative analysis of operational quality attributes, i.e. non-functional system characteristics. One of the description languages developed was *MetaH*. *MetaH* was the result of research carried out by the *Honeywell Advanced Technology Center* for use on avionics systems [100]. Derived from the development of *MetaH* and the need for an international standard for the aerospace industry, the *Society of Automotive Engineers* (SAE International) developed the *Architecture Analysis and Design Language* standard (AADL) in 2004 [167]. *AADL* is currently used to support modeling, analyses and validation of embedded software systems [100].

With all the advancements in model-based design and validation methodologies for large-scale and networked systems development, major challenges still existed. Challenges that led to noticeable issues like the crash of the European carrier rocket *Ariane 501* on the 4th of July 1996 because of software problems caused by flawed specifications. Due to assumptions encoded within the software for the physical system of its predecessor *Ariane 4* that were reused for *Ariane 5*, the rocket crashed because of lateral velocity limits after a chain of different events, including a failed conversion from a 64-bit floating point to 16-bit signed integer value that led to an arithmetic overflow [187]. Another example was the *Mars Climate Orbiter* that was lost in 1999 because English units were used within the software instead of metric units that were used during concept definition [40]. The first successful vehicle to explore another planet, the *Mars Pathfinder*, was deployed in 1996. The system worked as expected at first but exhibited infrequent cases of priority inversion. An unexpected interrupt that was sent to the systems bus caused a medium-priority communications task to be scheduled that blocked other important but low-priority tasks from being executed. The cause for the unexpected interrupts were wrong expectations for the performance of *Pathfinder’s* antenna which was not validated beforehand [105]. Other large-scale systems also exhibited major design problems. The development of the *Teledesic* satellite system, with an initial proposal requiring nine billion US dollars for funding, was abandoned after the need for major design specification changes had been encountered late in the design [311]. Meanwhile,

although the complexity for electronic systems was still growing with an average of 50% to 60% per year during the mid-1990s, the effectivity of design and validation methodologies was not. The average rate for effectivity growth of design methods at that time was 25%. This was referred to as *Systems Design Gap* [329], [30] and [313].

At the beginning of the new millennia, next generation commercial aircraft like the Airbus A380 were developed with more than 5000 ECUs in total and massive increase in overall functionality [314]. The amount of coupling between different aircraft systems increased as well. As an example, the amount of wiring for the Airbus A380 went up to more than 500 km [362]. Thus, the amount of wiring nearly doubled in comparison to previous aircraft developments such as the Boeing B747-400 with about 274 km of wiring [87]. Figure 2.11 depicts the trend in growth for the amount of lines of code and electronic equipment for avionics systems for different Airbus aircraft from 1970 to 2005.

First concepts for the A380 were already developed during the late 1980s and early 1990s, at the same time that Boeing started the development of the B777 [237]. The A380 became the first networked civil aircraft of Airbus [225]. Already during the 1990s, beginning with the Boeing B777, a novel integrated avionics systems architecture concept was used, known as *Integrated Modular Avionics* (IMA). In contrast to segregated and distributed avionics architectures that were used during the 1970s and federated system architectures prevalent during the 1980s, IMA are characterized by an integrated compound of configurable shared resources (hardware, operating system, application platform for subsystem applications and network) in conjunction with subsystem-specific hard- and software. IMA modules are partitioned according to different safety levels to host different segregated functions for each integrated subsystem [225] and [224].

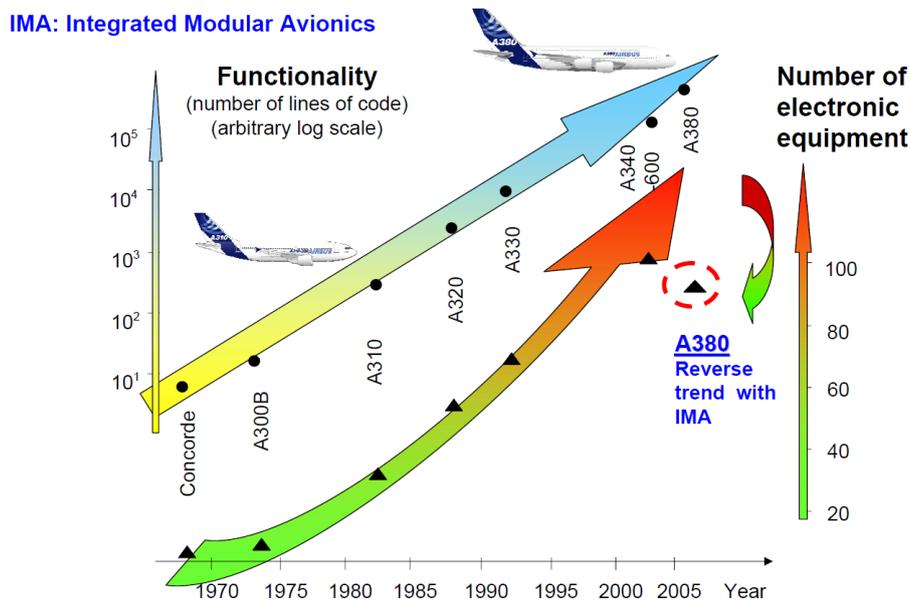


Figure 2.11 – Growth of the amount of lines of code and electronic equipment for avionics systems within different aircraft over time with decreasing growth due to the implementation of integrated modular avionics (IMA) (Source: Smyth and Roloff 2006 [335])

Because of growing complexity, Airbus went one step further for the design of a networked aircraft than Boeing during the development of the B777. As a result, Airbus

developed a deterministic and aircraft-wide network for the A380 [315] based on the networking technology *Avionics Full Duplex Switched Ethernet* (AFDX) [225] and [85]. This was done with the aid of validated specifications that were determined using *BONeS Designer* [315]. The overall system architecture of the A380 is composed of standardized line replaceable units (LRUs) and line replaceable modules (LRMs) connected by a common AFDX communication network [225] and [224]. With this, the trend of continuously growing number of electronic equipment was reversed as shown in Figure 2.11. Still, the amount of functionality was growing exponentially and, although qualified for highest safety levels, many aircraft subsystems were not integrated within IMA because of insufficient experience with the novel technology and concerns about safety aspects for flight critical systems. Moreover, the use of IMA did neither provide a concept to design optimal aircraft system architectures at aircraft level nor did it provide enough design flexibility for late specification changes [124].

Butz states [69], that already beginning in 2000, basic executable specifications were used for the development of the *Aircraft Condition Monitoring System* (ACMS) of the Airbus A380. The ACMS is a major aircraft system that has a wide variety of functions that range from aircraft performance monitoring, engine monitoring to trouble shooting. It has more than 20.000 requirements and, until then, was specified mainly with textual specifications. This led to ambiguities and inconsistencies within the specifications that resulted in errors discovered during implementation and test. While doing rework in order to remedy those errors, several problems were encountered. It was hard for the engineers to separate between specification and implementation errors because system specifications had a high amount of design uncertainty. Moreover, it was hard to separate between hardware and software errors. In the end, it took an average of three to five years to achieve a satisfactory level of system maturity. Due to huge testing efforts and many iteration cycles, customers were dissatisfied after development. After the introduction of executable specifications, specification errors have been detected during design. Thus, root causes for malfunctions were easy to trace, modifications were stable after bench tests, system suppliers were able to understand requirements unambiguously and 5000 pages of textual specifications were reduced to 150 executable statecharts. Most importantly, a sufficient level of system maturity was achieved [69].

Meanwhile, work on model-based design and validation for physical and structural aircraft systems progressed based on the developments created during the 1970s and 1980s. These “classical fields” of aircraft engineering designed, analyzed and validated systems under design at aircraft level thus resulting in good quality of specifications [315]. Examples include propulsion systems [162] and [80] or high-lift systems including landing flaps systems [228] and [229]. In 1998, the end of life for *BONeS Designer* was reached due to the discontinued commercial support of the programming language *LISP*, thus increasing the need for an adequate substitution [315]. At the same time, during the late 1990s, members from the department of electrical engineering and computer science of the *University of California* in Berkley founded the *Ptolemy Project*. The intention of the *Ptolemy Project* is the investigation of execution models and to study the application of heterogeneous modeling and simulation for the concurrent design of systems, with specific focus on embedded systems. This includes the mixed modeling of, for instance, analogue and digital electronics with hardware and software [194] and [111].

In response to the growing design and validation requirements for complex coupled and distributed systems, beginning in the early 2000s [311] and [315], the abstraction level for model-based systems design was raised to the system and mission level [323]. A company called *Mission Level Design Inc.* developed the multi-domain design tool *Mission Level Designer* (MLDesigner) that supported the mission level design approach as described, e.g. by Schorcht [323] as well as the import and usage of *BONeS Designer* systems. The architecture of the MLDesigner kernel is based on the *Ptolemy Project* and includes the *Ptolemy Tool Command Language* [222].

Using system and mission level design methodologies permitted to create and combine executable specifications (ES) with operational, maintenance and usability models to form virtual prototypes [222], [323], [324] and [311]. Examples include the development of large scale mobile satellite communication systems [323], Air Traffic Management Systems (ATM) [324], autonomous underwater vehicles (AUVs) [198], dynamic real-time systems like the *Terrain-masking Low-level Flight System* (TM-LLF) of the Airbus A400M [255] and avionics systems architecture optimization at aircraft level [124]. In the latter example, a potential overall wiring reduction of 68% could be demonstrated for networked integrated modular avionics systems [316] and [317].

Figure 2.12 depicts the time period from 1970 to today and provides an overview of design challenges, model-based design methodology development as well as representative system design, validation and verification tools.

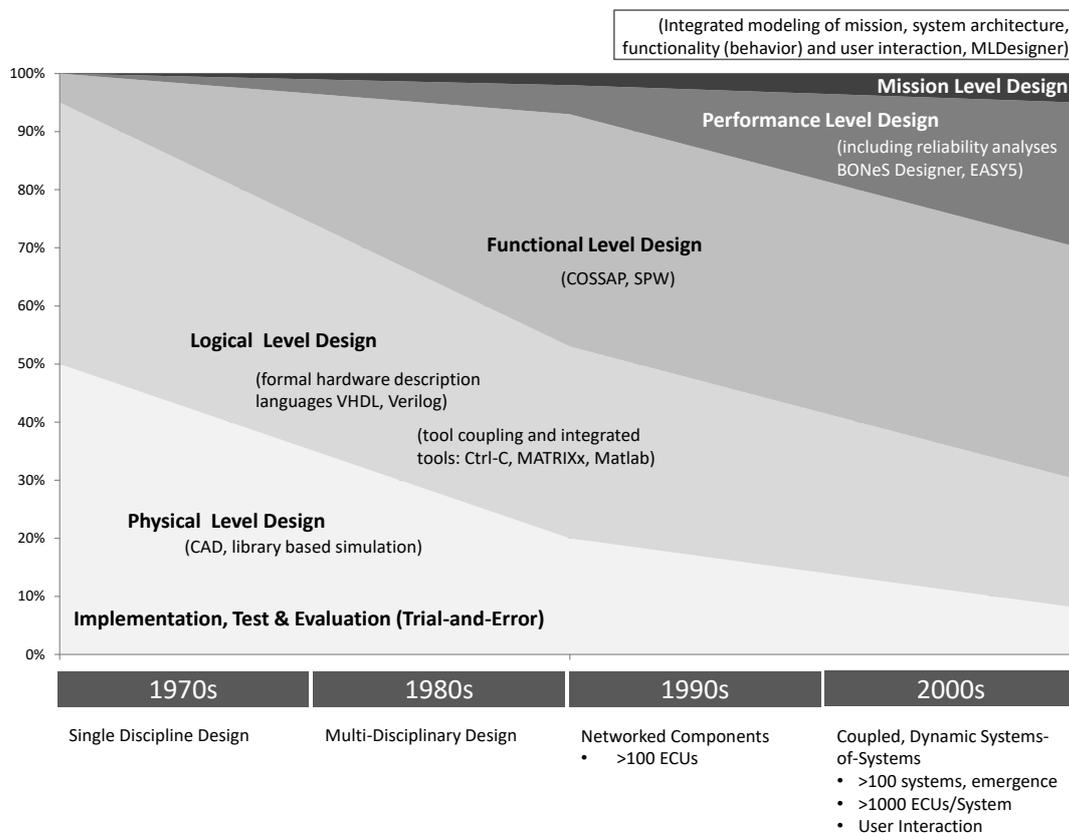


Figure 2.12 – Overview of the development of model-based system design, validation and verification methodologies for electronics together with associated design tools and design challenges from the 1970s to today (adapted from Salzwedel [310] and [312])

Currently, within the product development life cycle of civil aircraft, modeling languages for software and systems engineering, including the *Unified Modelling Language* (UML) or the *Systems Modeling Language* (SysML) developed by the *Object Management Group* (OMG), are being used to support aircraft software and hardware development [256]. Weikiens for example developed the *Systems Modeling Process* (SYSMOD) that uses SysML to model system requirements with regard to functional and physical architectures [365].

Different computer-aided software engineering (CASE) tools are employed in order to develop and document software and system architectures according to the *Department of Defense Architecture Framework* (DoDAF) [241] and to verify operational scenarios [67]. Beginning in the first decade of the new millennium, several organizations including aircraft manufacturer, subsystem supplier and universities formed a research cluster to foster the usage of model-based development methodologies and tools for aircraft development. As a result, the *Toolkit in Open Source for Critical Applications & Systems Development* (TOPCASED) was created [269], based on the open-source software development platform *Eclipse* [66]. TOPCASED is intended to be used for computer aided modeling of software and electronics systems in terms of implementation prototyping. The tool supports a variety of modeling languages including UML, SysML or AADL. It is also intended to be coupled with verification tools in order to check implementations against system specifications for correctness [269]. Another open-source tool for the implementation and verification of hard- and software for avionics using AADL models is the *Open Source AADL Tool Environment* (OSATE), which is also based on *Eclipse* [114] and [119].

The extension of integrated verification and test procedures including system simulation has been proposed to improve the quality for avionics software during development and realization [54]. For the development of safety critical software, e.g. for flight control systems, autopilot or braking systems the *Safety Critical Application Development Environment* (SCADE) was created (developed initially by *Telelogic*, now *Esterel Technologies*) to support formal functional verification activities. The design environment is qualified for the development of software according to *DO178B* and provides different levels of development, from aircraft level down to subsystem level. Integrated auto-coding abilities support the automated creation of certifiable software [303].

Certifiable software in terms of aircraft systems cannot be regarded as already certified yet. Instead, any auto-coded software still needs to pass the certification process for aircraft to provide adequate level of confidence according to *DO-178B* (cf. chapter 2.1). Figure 2.13 depicts, according to Whalen et al. [367], benefits and improvements achieved during software development for aircraft, spacecraft and other safety critical systems by using a model-based development and verification approach. It also shows the percentage of the amount of specifications that have been created and auto-coded based on computer models for respective subsystems.

However, functional specifications implemented with the tools shown in Figure 2.13 were not validated with executable models by design teams beforehand. Whalen therefore proposes design validation as necessary requirement before usage of *SCADE* or similar tools for verification [367].

The verification of functional and non-functional design aspects in one common model is currently not supported [41]. Moreover, as indicated by Whalen et al. [367],

functional specifications of coupled subsystems are not validated before the usage of *SCADE*, thus resulting in the inability to reduce design uncertainties early [367]. The usage of *SCADE* as a tool for systems design and validation has only been experimentally tested for selected software application developments. Again, one major reason is the preceding specification and validation process during early phases of the product development life cycle for aircraft that still heavily relies on the application of document cascades with traceability and requirements management tools in combination with manual validation [103] and [189].

Company	Product	Tools	Specified & Autocoded	Benefits Claimed
Airbus	A340	SCADE With Code Generator	<ul style="list-style-type: none"> • 70% Fly-by-wire Controls • 70% Automatic Flight Controls • 50% Display Computer • 40% Warning & Maint Computer 	<ul style="list-style-type: none"> • 20X Reduction in Errors • Reduced Time to Market
Eurocopter	EC-155/135 Autopilot	SCADE With Code Generator	<ul style="list-style-type: none"> • 90 % of Autopilot 	<ul style="list-style-type: none"> • 50% Reduction in Cycle Time
GE & Lockheed Martin	FADEDC Engine Controls	ADI Beacon	<ul style="list-style-type: none"> • Not Stated 	<ul style="list-style-type: none"> • Reduction in Errors • 50% Reduction in Cycle Time • Decreased Cost
Schneider Electric	Nuclear Power Plant Safety Control	SCADE With Code Generator	<ul style="list-style-type: none"> • 200,000 SLOC Auto Generated from 1,200 Design Views 	<ul style="list-style-type: none"> • 8X Reduction in Errors while Complexity Increased 4x
US Spaceware	DCX Rocket	MATRIXx	<ul style="list-style-type: none"> • Not Stated 	<ul style="list-style-type: none"> • 50-75% Reduction in Cost • Reduced Schedule & Risk
PSA	Electrical Management System	SCADE With Code Generator	<ul style="list-style-type: none"> • 50% SLOC Auto Generated 	<ul style="list-style-type: none"> • 60% Reduction in Cycle Time • 5X Reduction in Errors
CSEE Transport	Subway Signaling System	SCADE With Code Generator	<ul style="list-style-type: none"> • 80,000 C SLOC Auto Generated 	<ul style="list-style-type: none"> • Improved Productivity from 20 to 300 SLOC/day
Honeywell Commercial Aviation Systems	Primus Epic Flight Control System	MATLAB Simulink	<ul style="list-style-type: none"> • 60% Automatic Flight Controls 	<ul style="list-style-type: none"> • 5X Increase in Productivity • No Coding Errors • Received FAA Certification

Figure 2.13 – Examples and benefits for model-based development and verification (after requirements engineering) for safety critical software (Source: Whalen et al. 2007 [367])

Recently updated or newly developed quality and certification guidelines for software development of airborne systems like the *DO-178C* include the optional introduction of additional quality assurance methodologies during development such as model-based development techniques [277], [278], [280] and [279]. The described methodologies within these documents put most of the focus on how well the implementation reflects the specification [279] and [280]. Thus, system verification was improved rather than specification validation.

Written specifications formed at early design stages are partly enriched by graphical modeling notations like *UML* or *SysML* diagrams in order to improve specification quality and the following validation process [256]. But since these diagrams and graphical notations are embedded within written specifications, they are not executable by default, and thus cannot be validated through system execution, i.e. simulation. However, Wagner et al. state [121], that although in use for some years, there is no hard evidence that the use of UML within object-oriented design has been successful [121]. Moreover, experiences from model-driven systems development for telecommunication systems at *Motorola* showed, that the sole use of UML models as the base of model-driven development is not successful [364]. As part of ongoing research, members of the *Object Management Group* (OMG) under guidance of Stephen J. Mellor are developing a foundation to make UML models executable

[112]. This is referred to as *Executable UML* (xUML). Executable UML is designed to enable tools with the ability to build, verify, translate and execute UML-based models. A comprehensive introduction to xUML is provided by Mellor and Balcer [219]. Another approach to making UML models executable has been introduced by the OMG with the development of the *Foundational UML* (fUML) [249] and the *Action Language for Foundational UML* (Alf) [248].

In another work, Lemke [196] defined a development process based on UML/SysML to create model based requirements specifications. By means of behavioral simulation, he performed verification and test activities for the development of safety critical railway systems [196]. Weak points of this approach include the need for manual behavior evaluation as well as the concurrent definition of specification and test models in order to verify the system under design (SuD). Moreover, no concept validation by means of executable specification models is performed beforehand since Lemke's approach performs validation activities late during development, e.g. during implementation.

Executable validation models for avionics based on UML or SysML are, similar to the usage of *SCADE*, applied after specification development, mainly during systems realization or at more detailed subsystem development phases [357]. Examples include detailed timing validations for real-time critical avionics systems, e.g. to validate worst-case execution times (WCETs) for tasks with given hardware configurations [348] and [17]. Because these methods are used at late development stages, specifically on the right-hand side of the V-model, they cannot minimize product uncertainty early during development. As a result, these approaches are not able to minimize the overall risk of development.

Al-Homci [149] proposed to use an agent-based approach during aircraft systems development in order to perform validation, verification and test activities. Agents, often referred to as software-agents or bots, are a concept from artificial intelligence (AI). According to Woolridge [376], there is no uniform definition for the term "agent" since different fields of engineering and computer science use it [376]. Instead, there is a common understanding of what an agent represents. An agent is a computer or software entity that is situated in some environment and that is capable of autonomous, reactive and pro-active behavior in order to achieve specific goals or objectives [376] and [375].

In the case of system testing, agents are used as a representation of users and other stakeholders that perform different operations. Agents are often equipped with knowledge as well as rule bases and can sometimes learn new methods to react to unknown situations, i.e. situations that are not part of the original knowledge base. With regard to programming, one might compare the creation of agents to logical programming languages (declarative programming) such as *Prolog*. In Al-Homci's approach, the introduction of agent-based methods begins after specification development. Moreover, a specification formalization step is required before agents can be created. Therefore, this approach does not secure the validity of system specifications because it is applied too late during development, mainly for the purpose of testing. Moreover, if specifications are not validated before testing, an agent-based approach will also carry the same high amount of design uncertainty with associated development risk as currently used systems engineering methods.

In summary, taking a look at how and when executable design models are currently used during systems development, it can be noted that the major focus is set on the bottom and right-hand side of the V-Model, mainly on implementation, verification and test. Figure 2.14 illustrates the currently prevalent application of executable models during systems development. Here, the process of requirements engineering down to specification is carried out before the process of modeling and simulation. Requirements and system specifications, which constitute the baseline of development, are still created on paper and are validated manually. Hence it can be assumed that simulation models contain no more information than what has been provided by specifications. As a result, executable design models contain the same amount of design uncertainty as specifications. In consequence, model execution and evaluation will not be able to validate the design early in order to decrease product uncertainty.

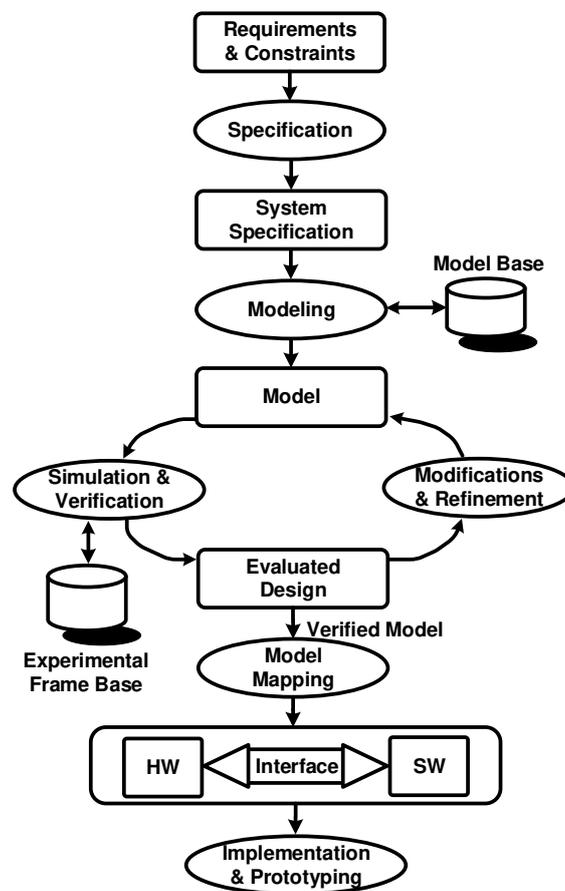


Figure 2.14 – Model-based design process with simulation and verification (Source: Rozenblit 2003 [299], based on [298])

2.3.4 Detailed Analysis - Design at Mission Level

In the previous chapters, classical system development approaches as well as existing model-based approaches for the development of complex systems, e.g. aircraft, were analyzed with regard to the minimization of the overall risk of development (cf. chapter 2.3.3). It was shown, that currently used approaches cannot be used in order to minimize the high risk of complex systems development. This is because no executable model of the overall system is developed during early design stages and

validation and re-design efforts are postponed to late development stages and are thus performed at the bottom or right-hand side of the V-model where expenditures for design changes are highest (cf. chapter 2.3.1 ff.).

It was also shown, that in contrast to classical development approaches, the mission level design approach was developed in order to shift validation activities to early development stages at the left-hand side of the V-model [323]. This chapter introduces the fundamentals behind the mission level design approach and evaluates it more closely. At the end of this sub-chapter, existing automation approaches in the context of model-based system engineering are analyzed with regard to the minimization of development risk.

2.3.4.1 Excursion - Scenario-based Requirements Analysis

During the 1990s, requirements engineering research was beginning to focus on methods with a goal or use case oriented perspective of the system under design (SuD) with regard to requirements analysis, specification and validation. As a result of this research, goals as well as context information for a SuD can be described with the aid of use cases and scenarios [210].

A major research project dealing with the usage of use cases and scenarios for the acquisition and validation of concept requirements was the “*Co-operative Requirements Engineering with Scenarios*” program that was funded by the *European Strategic Program on Research in Information Technology* (ESPRIT) [210]. In recent years, the application of use cases with associated scenarios has become an essential part of requirements analysis and concept validation [161]. A use case is a formal description that determines system requirements with the aid of different actors and multiple scenarios in order to achieve a specific goal or purpose (mission) [82]. According to Clymer [81], the mission level is the highest level of abstraction for a SuD, including functional and physical levels, and determines overall objectives of a system within its operational environment [81]. Use cases can be treated as collections for similar operational scenarios that include sequences of actions in accordance with rules that define links between actions. Actions combine timing, involved actors as well as objects that are being used [210]. The concept of use cases has also become part of the Unified Modeling Language (UML) and Systems Modeling Language (SysML) [365] as well as the concept of scenarios [247].

To include system security in the requirements engineering process, Sindre and Opdahl [330] and [331] proposed the creation and use of inverse use-cases and actors, i.e. misuse-cases and mis-actors, to supplement the mission profile of any SuD during early design phases. They developed description templates as well as misuse meta-models for the UML and SysML in order to be able to determine and document security driven system requirements [330] and [331]. These meta-models have since been extended, e.g. by Røstad [295], to differentiate distinctly between different kinds of internal and external system threats and vulnerabilities [295].

Since they are derived from use cases, scenarios also contain information about the intended operation of the SuD as well as about the intended environment (context) [345]. They define how users will interact with the SuD and its environment, thus helping system designers to gather and understand user needs. Moreover, they help users and designers to develop a common understanding of the SuD [270].

Each scenario represents a set of coherent and target-oriented interactions or steps between different internal and external actors. Actors can be human, e.g. system users, or non-human entities, e.g. system or system environment components. An interaction is characterized by definite trigger conditions, procedures and expected results [82]. Scenarios may also include boundary conditions [196].

Somé [336] divided the overall set of scenarios into subsets of so called positive and negative scenarios in order to determine and validate aspects of the SuD that either must be supported or else, need to be avoided [336]. One way to determine scenarios is to use statecharts [302], activity or sequence diagrams [365]. From an alternative perspective, scenarios may also be used in order to describe the envisioned functionality of the SuD with respect to its hardware and software [76]. An introduction to the scenario perspective on systems design can be found, for instance, in reference [75] or, in the case of a use case driven requirements engineering approach, e.g. for object oriented software, in references [173] and [365].

Carroll evaluated the process of scenario-based requirements analysis with regard to possible benefits for conceptual design and validation [76]. In summary, scenarios evoke early reflection about design issues by including end-user experience. Moreover, scenarios help to fight ambiguity in systems design since they minimize the number of possible interpretations. At the same time, they are flexible enough to be easily revised. Scenarios describe a SuD from the perspective of different stakeholders, including the intended users. As a result, scenarios support participation among all groups of stakeholders in order to arrive at a valid product design solution [76].

Figure 2.15 visualizes the relation between use case diagram (upper box), use case (oval node in upper box), scenarios (content of the use case node), and requirements specification (lower box). Sets of internal and/or external actors are related to each use case and entities of system context. For each use case, a set of scenarios can be derived according to the needs of stakeholders of the SuD. Use cases represent top-level requirements for the SuD with respect to stakeholder needs and system context while scenarios refine and concretize use cases in order to describe required services at system level. Thus, uses cases and services at system level can be used to create a conceptual model of the SuD as well as to derive and validate requirements specifications. This is because they describe what is expected of the SuD (for a set of specific goals and conditions) by whom and in what way (functional as well as quality properties). Single actions of scenarios can also constitute lower level use cases. This allows to hierarchically structure use cases in order to determine the entirety of requirements of a SuD [82] and [131].

However, different problems may occur in the context of scenario-based systems analysis and design. The foremost problem during the development of scenarios is that too many scenarios may be described for each single use case. This is due to the potentially high number of design variations and design alternatives and is sometimes referred to as scenario explosion [82]. Therefore, it is crucial to concentrate on the true intentions of stakeholders while providing an adequate level of abstraction, e.g. by consolidating different but similar design alternatives [82]. Other difficulties may occur in terms of scenario creation and management due to the disinterest of developers in detailed scenario specification [302]. A more detailed discussion on scenario related problems is provided in reference [82].

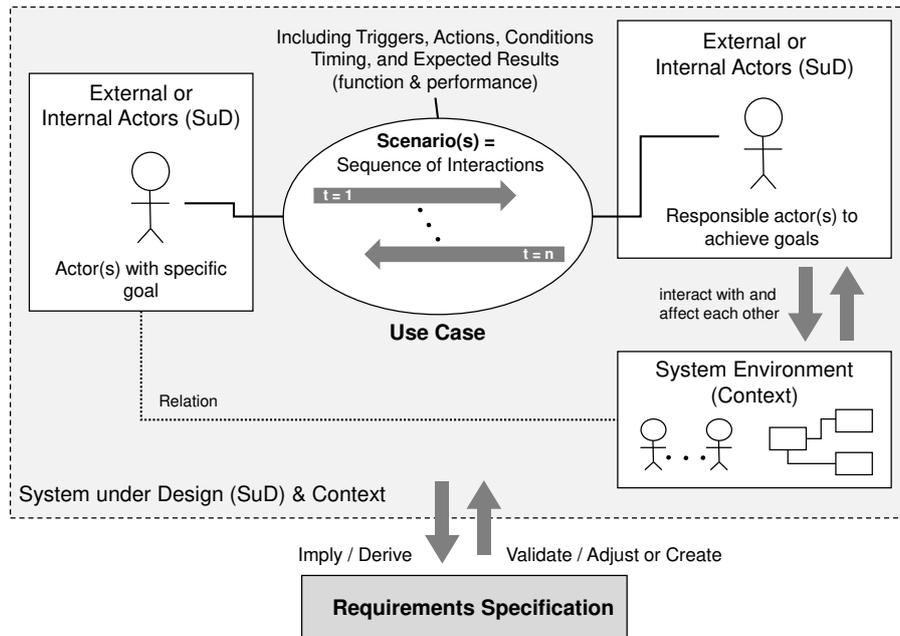


Figure 2.15 – Relation between use case diagrams (upper box), use cases (oval node in upper box), scenarios (content of the use case node), and requirements specifications

2.3.4.2 Mission-level Design Approach Evaluation

Before the introduction of the mission level approach, the system-level design approach was developed in order to shift design and validation efforts to early design stages [323] and [324]. Simulation is a key feature of system-level design [143] and it is used to analyze behavior and performance of the coupled overall system. By using a system-level design approach, it is possible to determine combined models for hardware and software that can be simulated in order to perform manual design space exploration [175].

Figure 2.16 depicts the process of system-level design. Beginning with textual or formal specifications, the system is modeled top-down in the form of an overall system model (grey box). This model includes system environment (context) to simulate external effects on the system, system architecture to account for limited resources of real-world systems and system functions, whereas functions are typically allocated to architecture components. Already existing and detailed component models at lower system levels (dashed line box downright) are abstracted and integrated within the overall system model in the form of macro models (bottom-up). Components that are not already present within the model database are developed in the form of abstract specification models that are enhanced with more details during later development stages (at lower left). This approach for system model design is also known as *middle-out design* [253] or *meet-in-the-middle design*, since an overall system model, which is developed top-down, integrates already existing detailed models of bottom-up developed components [50], [323] and [3].

A major advantage of re-using already existing system component models is that these have been validated and verified before. Re-using already validated and verified components for the design of higher level systems is also known in software engineering, often referred to as *Component-based Development* (CBD) [91]. Eventually, the overall system model is validated against the top-level specification [323].

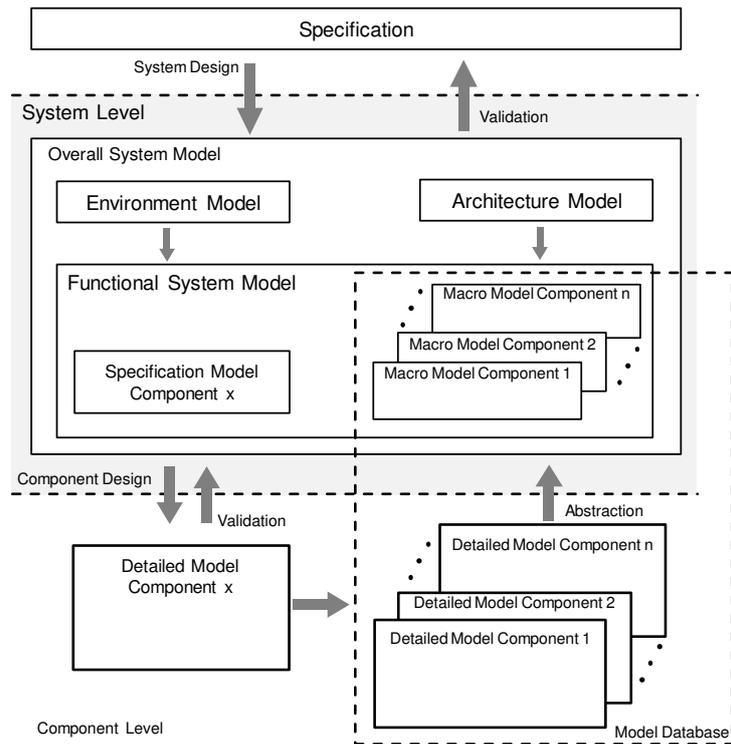


Figure 2.16 – System-level design approach (Source: translation of Schorcht 2000 [323])

In addition to the creation of environment, architecture and function models, two additional models are created, a driver model and an evaluation model. Both models are derived separately from the already existing specification. All five models are combined in order to generate an overall system simulation which is used to validate the behavior of the overall system. To do so, input sequences are inserted into the overall system model by the driver model during simulation. These sequences represent test cases for the system under design. On the other hand, the evaluation model is used to determine if the overall system model responds correctly to the set of given test cases [323]. Figure 2.17 depicts the general concept of overall system simulation that is used in order to validate overall system models.

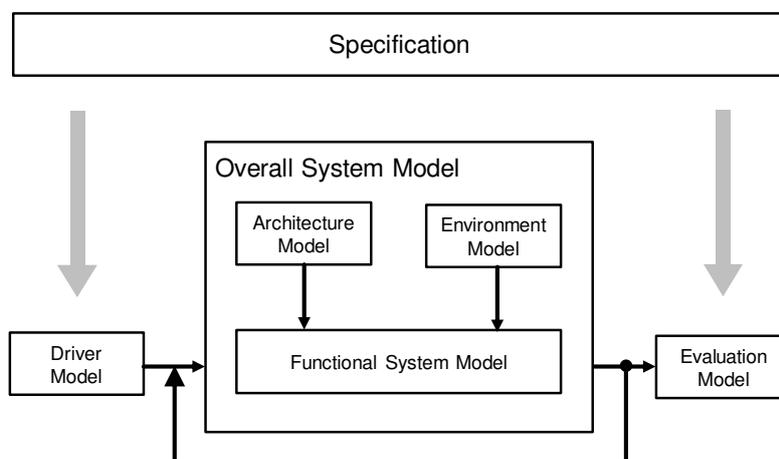


Figure 2.17 – Simulation-based validation at overall system-level (Source: translation of Schorcht 2000 [323])

Although system level design uses executable design models at system level and shifts validation activities to the left-hand side of the V-model, it is only applied after specification development and thus cannot minimize overall product uncertainty and development risk as early as possible. This is because the overall system model and simulation are still based on textual specifications that need to be developed during preceding design steps. Moreover, since dedicated models are developed in parallel for system stimulation, evaluation and the overall system, additional uncertainties may be introduced for each of the models, since the development process between each model is not directly coupled.

To overcome the described weaknesses of the system level design approach, Schorcht [323] introduced the mission-level design paradigm for the design of mobile communications systems. Mission level design is an extension of the system level design approach. Central aspect of this paradigm is the early formalization of operational requirements and system specifications through introduction of executable specifications and an additional level of abstraction, called mission level [323]. As depicted in Figure 2.18, mission models become part of the specification and represent a link between operational requirements and the overall system model. Similar to the approach of use case and scenario-based requirements analysis and requirements specification (cf. section 2.3.4.1), missions represent the formalized operational requirements of a system under design, i.e. its use cases. At the beginning of the mission-level design process, informal operational requirements exist that describe the complete set of applications for the system under design (SuD). These requirements are then described by means of use cases (missions) and become part of the system specification. Moreover, scenarios in the form of sequence diagrams can be derived from each mission. Based on the formalized operational requirements as well as informal non-functional requirements, an executable overall system model (cf. Figure 2.17) is developed at system level [323].

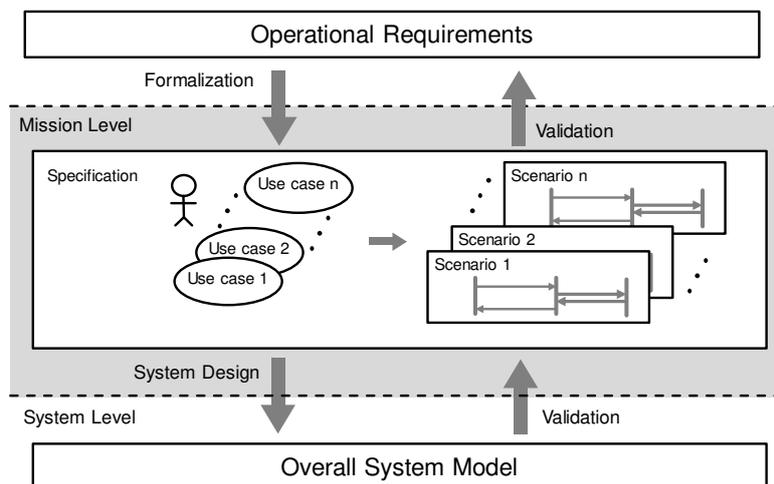


Figure 2.18 – System design at mission level (Source: translation of Schorcht 2000 [323])

According to Schorcht [323], the developed scenarios can be re-used in order to create driver and evaluation models that are used to validate the overall system model by means of simulation at overall system level. Thus, developed system specifications are validated against the operational requirements of the SuD. Moreover, Schorcht proposes, that the combined use of use cases and sequence diagrams can be used

for deriving test cases for implementations [323]. In the case of software testing, use cases, activity and sequence diagrams are already used in order to test software designs. Model-based test case definition can also be re-used for future developments with the same software product family in order to decrease development cost and time to market [290].

After the overall system specification model has been designed and validated for every mission, it is distributed to the different subsystem development teams in order to start the process of detailed development. Upon completion of subsystem development, abstracted macro models are integrated back into the overall system model to substitute the previously defined specification model components. The coupled overall system is executed again in order to verify that subsystem development has been successful with regard to top-level requirements [323]. Thus, integration problems that result from coupling effects between different subsystem entities can be discovered early during development by means of overall system simulation. Mission-level design shifts the focus of systems development and validation to early concept and specification phases in order to reduce product uncertainty earlier than traditional development approaches. Moreover, the use of an executable overall system specification permits early system optimization. With this, product uncertainty can be minimized early during development since design problems are already solved on the left-hand side of the V-model, when costs for re-designs are at a minimum. As a result, mission level design contributes to the minimization of the overall risk of development.

According to Liebezeit [198], using the mission-level design approach does not allow to gain generally valid assertions for the set of all possible system states of a SuD [198]. Instead, typical and intended scenarios for the application of the SuD can be validated and tested. This is wanted and due to the fact, that the design space of complex systems is too large to analyze all possible applications and interactions of the SuD. Therefore, the process of mission selection and development is most important during design in order to deliver a feasible subset for the range of possible applications of the SuD while satisfying all stakeholders. Limiting factors for the number of possible missions are the resulting simulation time as well as limitations due to the required manual evaluation [198].

Schorcht's work about mission-level design [323] provides an approach for systems specification and validation based on detailed computer models. Although Schorcht determined the general aspects of a mission level design approach, he did not determine a detailed description of a workflow that determines how to perform each step of the overall design and validation process. In general, it is possible to automate system specification validation by means of the scenarios developed at mission level. To be able to do so, it is essential to provide machine-readable scenario descriptions in order to execute driver and evaluation models. However, the mission level design approach as described by Schorcht [323] does not provide integrated means to automate design, optimization and validation processes as part of an iterative development process. Moreover, no detailed descriptions for the modeling process of mission and scenario models were provided. Schorcht developed no basic mission model components or libraries that could be used as part of an integrated design and validation environment in order to support the mission level design approach. The creation and application of combined driver and evaluation models in the context

of system validation has also not been covered by Schorcht [323]. This leaves users of the methodology with the challenge of how to develop, implement, interface and apply mission, driver and evaluation models.

Although non-functional requirements are part of the overall system model (executable system specification), they are currently not necessarily included within the mission and scenario model for system validation. In addition, Schorcht provided no detailed description on how the combined behavior and performance of the overall system model can be validated against the mission model during simulation. Thus, driver and evaluation models can currently only be used for validation of operational or behavioral requirements for the SuD but not for the validation of top-level performance requirements. These requirements have to be validated manually during overall system simulation, which also limits the possible number of missions [198]. As shown in references [365] and [132], operational scenarios in the form of activity or sequence diagrams can also be extended by including non-functional requirements, e.g. timing requirements.

This raises the following questions to be answered in the context of improving the mission level design approach: How to perform the transition between concept design and mission level design? What are the steps to be performed beginning with mission model development through to overall system model development? What kind of information does the mission model need to provide? How are models to be coupled? How to develop a combined driver and evaluation model for functional and non-functional design properties? Are missions and scenarios to be simulated and validated sequentially, in parallel or at random? How to treat missions and scenarios that exclude the parallel execution of another mission or a scenario because of specific conditions and constraints? How to perform the validation of different missions and scenarios that depend on or require each other? How can the process of design and validation be automated?

2.3.4.3 The System Design Environment MLDesigner

MLDesigner (MLD) is a system-level modeling and simulation environment available for GNU/Linux and Microsoft Windows operating systems and supports the mission level design approach as described by Schorcht [323]. The product is developed by MLDesign Technologies, Inc. Palo Alto, CA, USA and Mission Level Design GmbH, Ilmenau, Germany. MLDesigner provides an integrated design environment (IDE) with graphical user interface (GUI), as depicted within Figure 2.19, to build models and execute simulations. Moreover, external simulations can be generated in the form of compiled code that can be executed externally. The following information has been compiled from the official MLDesigner manual [222].

MLD includes a multi-domain simulator that supports modeling and execution of different computation domains such as Discrete Event (DE), Synchronous Data Flow (SDF), Dynamic Data Flow (DDF) and, among others, a finite state machine (FSM) domain which is based on statecharts that were developed by Harel [156]. The FSM domain can be used similar to FSM models of the Unified Modeling Language (UML). For each domain, MLDesigner automatically performs formal verification for every model created (model checking). Technical applications range from design and optimization of complex avionics systems for aircraft, global satellite communication systems, networked automobile electronics, as well as software and hardware for

embedded systems. Moreover, complex processes can be modeled and simulated. Simulation models can combine multiple domains to represent heterogeneous system architectures and can be created and simulated on a personal computer or laptop. It is also possible to run distributed simulations on a cluster of workstations. Multiple users can work in teams by application of a version control system, e.g. open source solutions like Concurrent Versions System (CVS) and Apache Subversion (SVN). Each model is stored in a hierarchy of structured files using the XML (eXtensible Markup Language) standard.

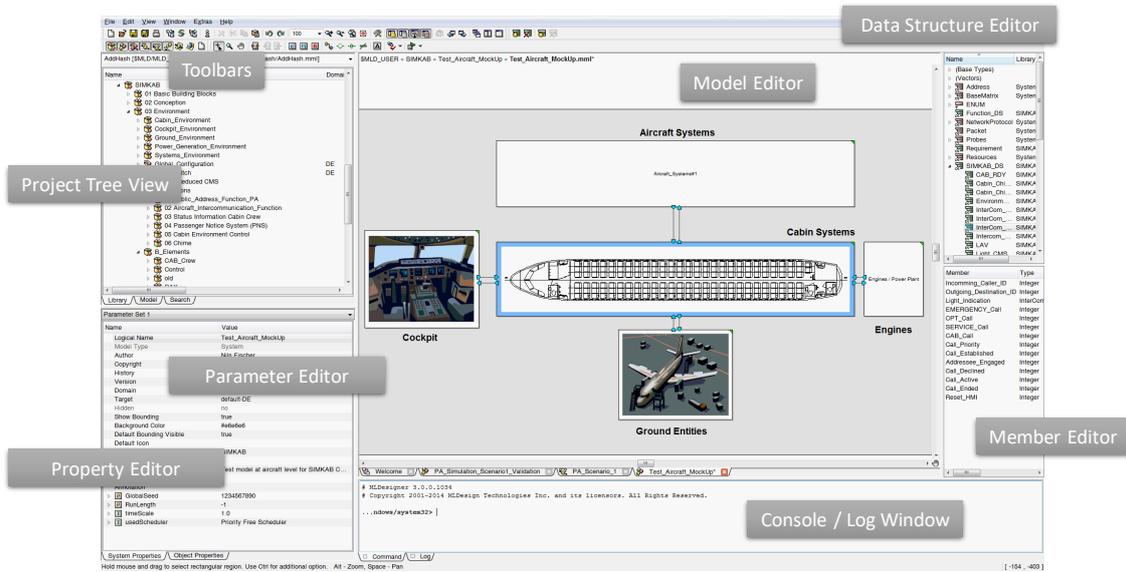


Figure 2.19 – Graphical user interface of MLDesigner 3.0 with annotations

Models are built hierarchically, each incorporating a series of building blocks of library elements, sub-modules and relations in-between as depicted in Figure 2.20. The functionality of MLDesigner modules may be specified by a hierarchical block diagram composed of predefined model library elements shipped with MLDesigner, finite state machines, modules defined in C/C++ language (called Primitives) or by combinations of the different types. Data is exchanged through ports (triangles) and edges that connect ports. Edges are used to model data flow as well as control flow. The highest level in each model hierarchy is called system. Systems can be simulated according to the specified model of computation, called domain. Models are either constructed by using a graphical editor with predefined components or by direct ANSI C/C++ source code injection from custom designed elements. MLDesigner’s scheduler and consistency checker automatically identifies syntax errors during system execution.

MLDesigner supports basic data types such as integer, complex data types, arbitrary structured data types, enumerated types, as well as user-defined data types. Data types and structures are managed in a dedicated data structure and member editor. This editor can be accessed from the MLDesigner main GUI (see Figure 2.19). A tree in the upper right region shows the hierarchy of data structures that can be accessed via a drop-down menu, e.g. to select or change the respective data structure. Input fields change dynamically once a type is select from the drop-down list.

Every library, model component and data structure can be annotated by using description and annotation parameter fields. With that, texts from e.g. requirement

documents can be directly included within models. The automated documentation generation function of MLDesigner can be used to create Hypertext Markup Language-based (HTML) documentation for simulation models. Documentation includes all information on a specific library or model, including model interactions, data structures, requirement texts and annotations.

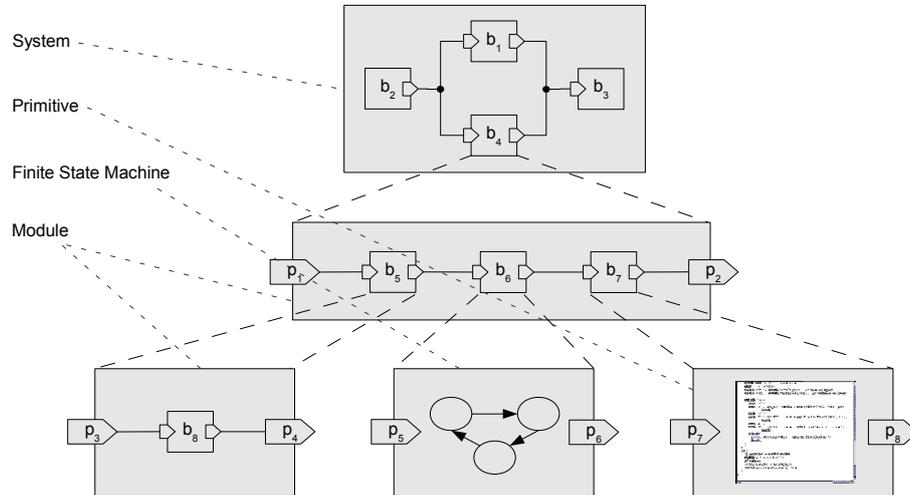


Figure 2.20 – Hierarchical modeling principle of MLDesigner (Source: translation of Baumann 2009 [30])

In terms of interactively controlling and monitoring simulations, MLDesigner offers pre-configured GUI library elements. These can be operated standalone or in combination to visualize data during simulation. In addition, so called probes can be used to observe data ports of models during simulation. Custom simulation GUIs can also be created by using predefined Primitives and Tcl/Tk source code files. More information about MLDesigner can be found on-line or in reference [222].

2.3.5 Automation Approaches for MBSE

In this chapter, related work within the scope of model-based systems engineering (MBSE) automation is analyzed with specific focus on mission level and use-case-driven design approaches. Each approach is evaluated with regard to the early minimization of product uncertainty and thus, the minimization of the overall risk of development. Existing automation approaches are arranged from validation to verification, test, model generation and architecture optimization.

2.3.5.1 Validation Automation

In the context of requirements engineering for large, real-time unmanned weapon systems, research within the *Software Requirements Engineering Program* (SREP) led to the development of the *Software Requirements Engineering Methodology* (SREM). SREM includes the description of a systems engineering process with associated software tools and is used to perform specification development and independent validation based on a formal requirements engineering language and system simulation [8] and [9]. Major objectives of the research behind SREM were the reduction errors in software requirements and to provide more automation in terms of validation [9].

A key concept of SREM is to specify software requirements in the form of sequences of processing steps called *paths* [8]. Each Step includes descriptions of specific input stimuli or expected output response for the system under design (SuD). Path models can be compared to highly structured finite state machines [10]. The terms stimulus, stimuli and expected response are also used as part of the *UML Profile for Schedulability, Performance, and TimeTM*. Here, stimuli are used to model dynamic situations with cause and effect relationships for systems with real-time characteristics. For instance, a stimulus could be a message of a sender to a receiver, e.g. as part of a sequence diagram [247].

In addition to functional requirements, performance requirements can be determined in the form of variables that can be measured along paths. At specific points of each path, so called validation points are determined. Validation points are used to acquire measurement data during system simulation and are used in order to test whether arriving data is valid with regard to path specifications. Paths that are initiated by one specific type of stimulus and use the same interfaces are integrated into so called *requirements networks* (R-Nets) [8]. Figure 2.21 depicts an example path analysis structure created by Boehm that illustrates the concept of validation within path sequences [47]. Validation points that receive stimuli or response messages are placed close to interfaces. Between validation points, different processing steps are performed.

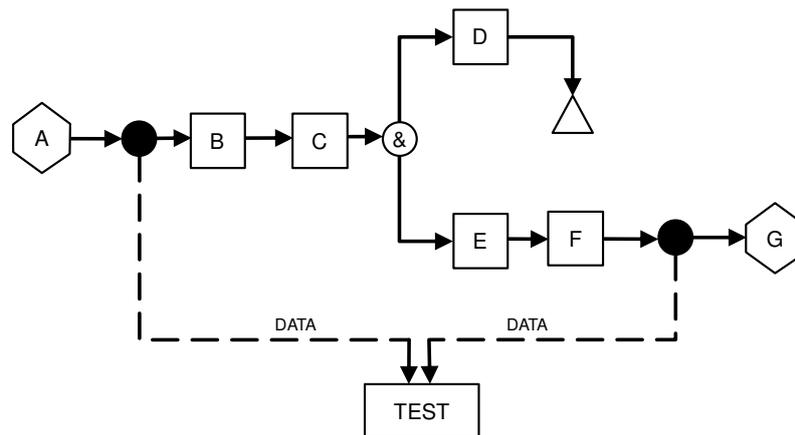


Figure 2.21 – Path analysis structure example of the *Software Requirements Engineering Methodology* with different functional input/output processing blocks (squares), interfaces (hexagons) and validation points (dark circles) close to interfaces (Source: Boehm 1984 [47], based on the development by Alford [8])

To be consistent with the state machine model, only one R-Net can be active at a time during simulation [10]. This makes it impossible to validate different requirements and paths concurrently. In addition, no overall system model exists that couples function, architecture and system environment.

Within the field of component-based software development for real-time systems, Fleisch developed an approach to dynamically validate system specifications by means of simulation models [133]. Figure 2.22 depicts the automated validation process developed by Fleisch. By analyzing software design quality as well as dynamic software testing methods including white-box and black-box testing, he concluded that test methods are applied too late during development in order to raise

software specification quality. Thus, he developed an automated method for early simulation-based validation for real-time software [133].

Similar to Schorcht [323], Fleisch's approach [133] starts with the specification of operational user requirements (missions) in the form of informal textual use cases. For this purpose, Fleisch provided a template in tabular form. Use cases are transformed into different application scenarios based on a machine-readable notation. This transformation is realized with the aid of extended UML sequence diagrams, i.e. sequence diagrams that are extended with conditions and timing. Based on the use case-driven requirements specification, a component-based simulation model is developed. This model is created with the aid of already existing model components from a software model database. During a third step, test cases are derived manually from all sequence diagrams in the form of stimuli sequences. Stimuli sequences are also referred to as comparison sequence diagrams. A stimulus sequence has a specific starting time and consists of timed and sequential messages that address different actors of the SuD. All messages are injected into the simulation model by directly addressing associated interfaces and components of the software under development. Thus, Fleisch's method can be regarded a white-box testing approach. During system simulation, reactions of the system model to the different stimuli are recorded in the form response sequences with time stamps, called trace sequence diagrams. After simulation, a conformity check is performed between the formalized stimuli sequences and the recorded system behavior sequences [132] and [133].

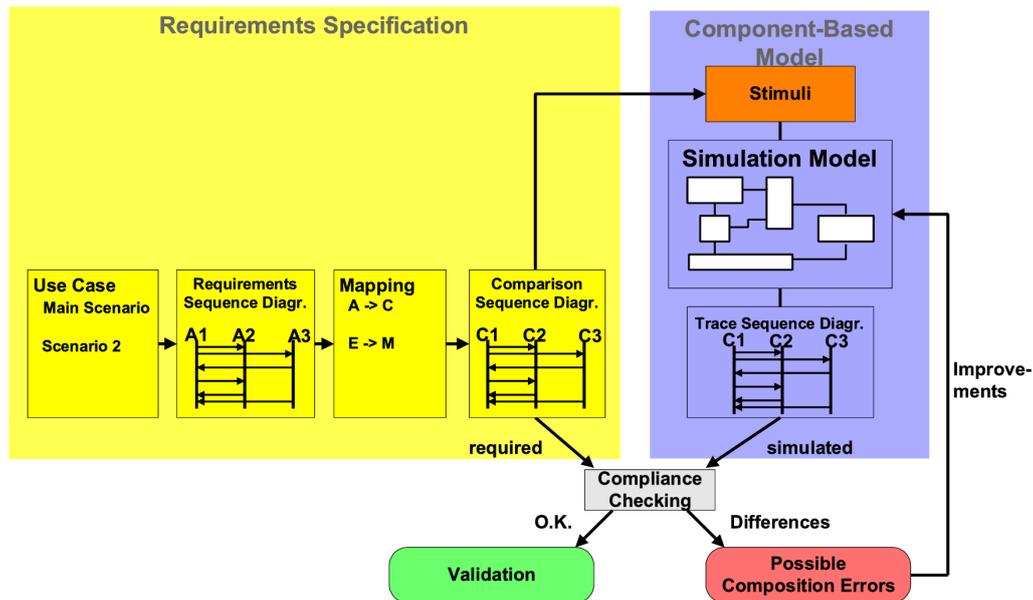


Figure 2.22 – Automated software requirements specification validation process (Source: Fleisch 1999 [132])

For validation purposes, every scenario is simulated and validated at least once. This is either done in sequence (stimuli sequences are executed one after another) or overlaid (several stimuli sequences are executed together). However, Fleisch only used and evaluated sequential scenario execution for the purpose of design validation and did not provide a description of how to structure and compose concurrent stimuli scenarios. Thus, preconditions and interactions defined by specific scenarios may lead to contradictions during the overlaid and parallel execution of different

stimuli sequences. Fleisch determined another possibility for automated validation based on randomized stimuli with stochastic distributed data. According to Fleisch [133], this second approach would lead to extremely long simulation times in the magnitude of years when used for large and complex systems. He therefore dismissed this alternative approach as a way to find a feasible solution for validation automation [133].

Fleisch's approach [133] requires an existing database with verified software components in order to create a simulation model. For the conceptual design of complex systems such as aircraft, not all subsystems or components may already be available in the form of detailed model components. Since the applied test method is based on a white-box strategy, it is essential to provide detailed descriptions of processing steps for the SuD. Thus, a profound knowledge of the intended system architecture and implementation-specific aspects for the software under development are required during sequence diagram creation. Moreover, an extensive amount of manual work has to be performed in order to derive comparison sequence diagrams from requirements sequence diagrams. This is due to the necessary mapping of actors and events of requirements sequence diagrams to components and messages within the overall simulation model. Figure 2.23 depicts the manual mapping process for the example of an electrical passenger window for cars. Fleisch was not able to automate this process. He also stated [132], that it is less suitable for the design of continuous feedback control systems [132]. As a result, the proposed automated validation process by Fleisch cannot be used to minimize product uncertainty of complex and dynamic systems, e.g. aircraft, early during design and the overall risk of development cannot be minimized.

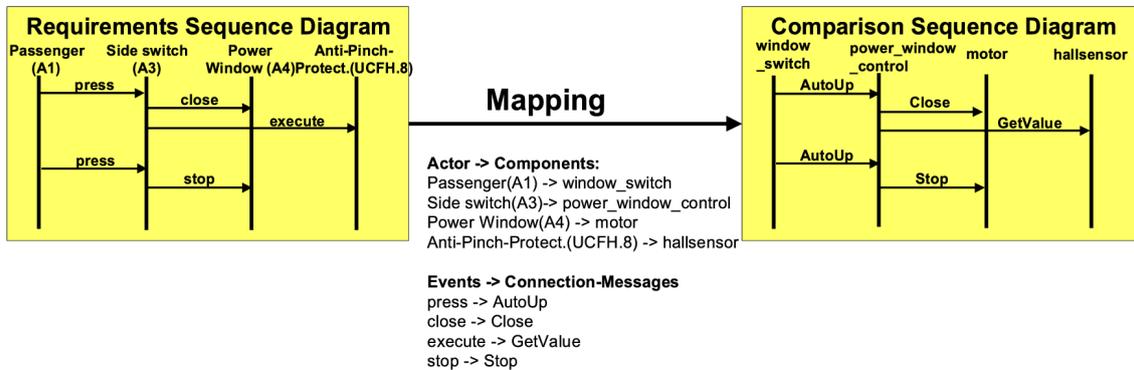


Figure 2.23 – Manual mapping process from requirements sequence diagrams to comparison sequence diagrams for the example of an electrical passenger window for cars (Source: Fleisch 1999 [132])

2.3.5.2 Verification Automation

Pacholik [254] compared and developed concepts for automated verification of time constraint functional system properties within executable specifications. Two general methods were described by Pacholik [254]. Firstly, by using a transformation approach for discrete event models so that existing verifications methods and related tools can be used for complete and formal verification of the SuD. For this transformation-based approach, the verification problem is mapped into a mathematical model that can be automatically and formally verified. Secondly, a dynamic verification method for system properties was developed that uses scenario-based

simulation. This is achieved by using a so-called assertion-based approach in which required system properties (assertions) are defined in the form of temporal logic formulas that can be evaluated by computer algorithms and by means of co-simulation. During simulation, it is possible to decide dynamically if the SuD provides required properties as specified [254]. By using his approach, Pacholik is able to verify that system properties defined during detailed systems development and implementation are in accordance with existing product or system specifications. However, no automation is provided for model-based validation during early design stages at the upper left-hand side of the V-model. Thus, it is not possible to decrease product uncertainty and the related risk of development early during system design.

2.3.5.3 Test Automation

Related work has also been done in the field of agile software development and model-based software testing in particular. As part of agile software development processes, “*user stories*” have been used for both, requirement definition and testing. In the context of this work, a user story can be compared to the definition of use case-related scenarios. More information on the application of user stories can be found e.g. in reference [83]. In his work, Lemke developed a SysML-based method and process model in order to create model-based requirements specifications, that uses activity diagrams to substitute use case diagrams [196].

In the field of dynamic software analysis techniques, different black box approaches for system testing are available. Dynamic analysis techniques test software by execution of test objects on a computer [337]. Spillner et al. [337] describe the use of state machines as well as use cases for testing. Test cases are described in the form of sets of preconditions, inputs, expected results and postconditions. A test is considered successful, if a test case has been simulated as specified. In the case of applying use cases for testing, concrete input data and results cannot be directly derived and need to be determined individually for each test case [337].

Model-based testing uses computer models for test automation as well as for the automation of test case generation. Moreover, model-based testing processes can be driven by an overall system model, an independent test model or both [292]. Schieferdecker [321] has described different variants of model-based testing approaches, including an approach to use a combination of system and test models. As part of the Unified Modeling Language (UML), a fundamental testing profile has been developed, UML Testing Profile (UTP) or UML 2.0 Testing Profile (U2TP) [292]. The use of test cases as part of a UML-based systems development process has also been covered by UML related literature, e.g. by Weilkiens [365].

Roßner et al. [292] describe the application of executable test case compositions in the form of activity diagrams for regular sequences of test cases. However, during automated path coverage testing, invalid paths may be executed [292]. Utting and Legiard [355] investigated the usage of finite state machines and UML transition-based models for testing in order to be able to make test cases executable on a system under test (SUT). In order to do so, they proposed the use of manually written software adapters to bridge the gap between test cases and SUT or to use generated test scripts [355]. The use of UML diagrams for test case generation has also been proposed by Kundu and Samanta [186], Kansomkeat et al. [177] and Linzhang et al. [202].

Lehmann [195] developed an approach called *Time Partition Testing* (TPT), which is intended for testing of continuous behavior of embedded systems. As part of his approach, test procedures for reactive systems are determined with statechart-based scenarios that describe sets of system input signals together with expected system responses [195]. A test tool called TPT has been developed that is based on statechart test models to support Time Partition Testing. Bringmann and Krämer [57] proposed the use of TPT for the development of automotive embedded devices [57]. Scenarios have also been successfully used in terms of model-based testing by Arnold et al. in reference [21].

As is the case with Model-based verification automation, automated model-based testing may successfully contribute to finding implementation errors as well as design errors. But since the development process has already arrived at the right-hand side of the V-model, design errors that stem from flawed specifications, due to a high amount of product uncertainty, are discovered too late during development. Thus, automated model-based testing cannot be used to minimize the overall risk of development during early design stages.

2.3.5.4 Automated Model Generation and Optimization

Baumann [30] used model component annotations for the partitioning of sets of existing model components. This was done to automate the generation of model variants as part of the mission level design approach. The intention of his development was to provide system developers with the ability to perform early performance analyzes for different networked system architecture designs that are based on the same set of functional requirements.

After requirements analysis, a mission model is developed. Simultaneously, a functional model and an architecture model are developed. A functional or behavioral model is used to describe ideal functional system requirements, i.e. system functions without performance aspects. An architecture model combines functional model components with available resources, data communication components as well as task execution-related components. Components of architecture models can be developed by using a bottom-up approach. All three models need to be validated separately against top-level system requirements. After successful validation, functional model components can automatically be mapped to components of the architecture model (partitioning) [30]. This process is performed based on partitioning information (annotations) provided within the functional model and by using a stand-alone model generator program that has been developed by Rath [282] and [30]. By using an automated model component mapping approach, it is possible to create different architecture models with different mappings of functional model components in order to evaluate the impact of different model setups on overall system performance.

In 2007, Fischer [124] and [316] extended Baumann's automated mapping to optimize avionics system architectures. In his work, Fischer used a set of different models and simulation steps, from behavioral models through to system constraint models, architecture optimization models as well as architecture model generation. With this, it was possible to determine an optimized system architecture for avionics with a potential reduction for overall cabling of up to 68% [124]. However, both approaches can only be used after successful concept design, requirements specification and validation. Moreover, modeling and simulation steps described by Baumann [30]

and Fischer [124] may need to be performed repeatedly and automatically as part of a system development process. Thus, both approaches, taken in isolation, cannot be used to minimize product uncertainty early, since they do not provide the means for the combined automation of validation and optimization activities during early stages of a mission level design approach.

2.3.5.5 Workflows

In economics, workflows are used to orchestrate and automate processes. A workflow can be defined as a recurring process that is based on the division of labor. More importantly, workflows manage tasks, processing units and the network of relations in-between, including operation sequences and data flow [320]. In a broader sense, a workflow could also be described in terms of a flowchart with a repeatable set of sequenced, structured and orchestrated tasks with intermediate goals, expected results, data exchange, limited resources, quality objectives and constraints in order to achieve one or more overall goals.

In science, especially in the field of grid computing [382], so called scientific workflows are used to support scientific research by accessing, controlling and orchestrating remote and distributed data sets in collaboration with remote computational resources [93]. A scientific workflow consists of a structured process graph that is used to automate the execution of a set of different scientific processes, i.e. to perform sets of complex and data intense computations that include different data processing tasks and dependencies [217] and [101]. By using scientific workflow management tools like the free software *Kepler*, scientists can integrate and orchestrate existing components for data acquisition, transformation and analysis into larger systems that are used for an even more comprehensive analysis, e.g. of large data sets [388]. An example scientific workflow with Kepler is shown in Figure 2.24. Other forms to describe scientific workflows include the usage of Petri nets and activity diagrams of the Unified Modeling Language (UML) [101]. Since scientific workflows are designed to structure and distribute processes for data acquisition, data preparation and processing, they cannot be used for the automation of design validation and optimization processes as part of a mission level design approach.

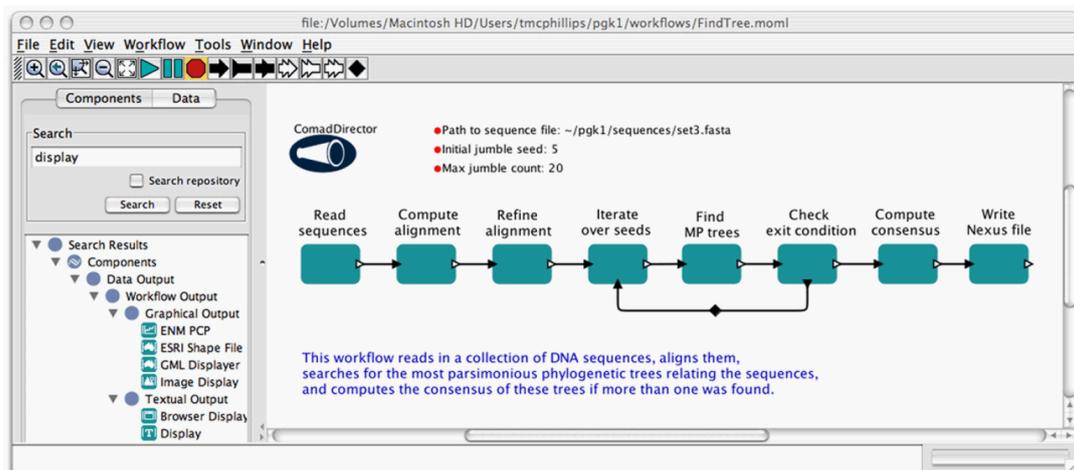


Figure 2.24 – Scientific workflow example in Kepler with different tasks (rectangular nodes), configuration parameters (red dots), and directed edges used for modeling of control and data flow (Source: McPhillips et al. 2009 [217])

Workflows are also created and applied in computer science, similar to the description above, in order to orchestrate and execute different processes with specific goals, including control flow and data flow [356]. In this context, activity diagrams of the UML [104], [28] and [267] or Systems Modeling Language (SysML) [257] have been proposed to function as specification language. Comprehensive work in the context of activity diagram-based workflow development has been carried out by van der Aalst, Hofstede et al. [356].

2.4 Detailed Objectives of This Work

Based on the previous analysis of the problem to be solved and the evaluation of existing solution approaches and related work, this chapter determines what has to be done in order to improve the quality of specifications for the development of complex systems. Detailed objectives and contributions for this work are elaborated together with requirements and boundary conditions for the proposed solution.

2.4.1 Minimum Risk Model-Based Systems Engineering

As elaborated within the previous chapters, good quality of specifications can only be achieved, if these include as little product uncertainty as possible and thus represent “*the right system*” (cf. chapter 2.3.1). To be able to minimize the amount of product uncertainty early during system development, specifications need to be validated before commitment in the presence of bounded and statistical uncertainties, beginning with concept design. System designers must therefore be able to validate, that the intended application or mission of the product will be successful with regard to all stakeholders, i.e. the design problem is solved in the best possible way by the system under design. Moreover, the process of validation needs to ensure, that the intended system functionality can be achieved in conjunction with performance requirements, given constraints and, if applicable, re-used system components from previous development projects. In the case of highly configurable aircraft systems like cabin management systems, early design validation needs to include system configuration and customization as well as usability aspects (cf. chapter 1.1). In parallel early design space exploration needs to be performed in order to develop an optimal solution at overall system level.

In doing so, product uncertainties are resolved early and will not be carried through the rest of the development process. Thus, integration problems can be solved during design, when cost are at a minimum, instead of during integration and operation, when cost are at a maximum. As a result, the overall risk of development can be minimized. As proposed by the *International Council on Systems Engineering* (INCOSE) [144], [112] as well as the *National Defense Industrial Association* (NDIA) [34], a model-based systems engineering approach is developed in this work in order to improve the quality of system specifications early during development. To do so, the usage of validated executable specifications and virtual prototypes is proposed (cf. last section of chapter 2.3.2), based on the application of a mission level design approach:

- as an improvement of available model-based systems engineering methods
- to find a common design description language between engineering disciplines

- to be able to perform early automated design validation and testing

The feasibility and impact of model-based development approaches in general has been shown in different studies and research projects from different fields of industry, e.g. in references [341], [311], [218], [364], [122], [367], [223], [123], [317], [387], [182], [215], [181], [72], [62] and [26]. These studies indicate, that using a model-based development approach during design, development and test can reduce overall development time and costs through early design uncertainty reduction of at least 10% to 30%. Experiences from the aviation industry show that a considerable productivity gain of more than two orders of magnitude can be achieved by application of formal and model-based engineering methods [69].

It is the objective of this work to develop a model- and simulation-based systems engineering methodology to improve the quality of complex system specifications by reducing product uncertainties early at the left-hand side of the V-Model development process. In order to do so, the process of validation is improved, including the provision of the means for automated validation and optimization. The aim of the proposed method is to significantly reduce the amount of late and cost intensive design and specification changes in order to minimize the overall risk of development described in chapter 2.2. The proposed model- and simulation-based system development method is referred to as *Minimum Risk Model-based Systems Engineering* (MR-MBSE). In the context of this work, MR-MBSE is defined as follows:

Definition 4 *A **Minimum Risk Model-based Systems Engineering** (MR-MBSE) process applies formal computer models as central objects of the overall system development process. It minimizes the overall risk of development, i.e. it maximizes the probability for the successful completion of system development projects by providing the optimal solution to a given design problem without exceeding given time and cost budgets. This is done by shifting most design and validation efforts to early development stages at the left-hand side of the V-model in order to minimize product uncertainty early during design, when expenditures for design changes are a minimum. In order to achieve MR-MBSE, all development decisions are validated early against the services of a product at mission level by using validated executable specifications (VES) and virtual prototypes (VP) that are used and updated from concept development through to the end of life of a product.*

In addition, as part of the development of an MR-MBSE method, so called executable workflows and simulation sets are developed in this work in order to support the creation of executable processes and integrated executable models that can contain sets of independent simulation models, e.g. for automated design validation and optimization. Both concepts are elaborated within the following sub-chapter. The proposed MR-MBSE method is developed in conjunction with an associated plug-and-play capable system design and validation environment with predefined library components.

The developed solutions are demonstrated and evaluated by using examples for civil avionics and cabin systems development from different research projects. These research projects were partly sponsored by the German *Federal Ministry for Economic Affairs and Energy* (BMWi) (formerly known as *Federal Ministry of Economics and*

Technology) under grant numbers 20K0702A, 20K0806A, 20K0805A and 20K0805R. Different issues that are not specifically addressed by this work are listed below:

- Automated verification and test of implementations.
- Version management to support teamwork at different stages of development.
- Integration of the developed methodology with requirement management tools.
- Code generation from executable specifications.
- Automated model generation.
- Automation of safety analyses.

2.4.2 Executable Workflows and Simulation Sets

In general, executable workflows are similar to flowcharts, functional flow block diagrams, control flow diagrams or activity diagrams of the Unified Modeling Language (UML). Within this work, a concept and implementation for executable workflows and simulation sets is developed for the proposed MR-MBSE method. This is done in order to enable automated and repetitive execution of different design, optimization and validation steps as part of a phase-oriented and iterative system development processes, e.g. the V-model. Repeated design, validation, and test steps are also necessary during later phases of the product life-cycle. Usually, when a product has successfully achieved entry into service, new application and performance goals arise from different stakeholders. As a result, these goals need to be integrated within the existing design before the overall design is validated again. The ability to develop executable workflows and simulation sets is a prerequisite for the development of a Minimum Risk Model-based Systems Engineering methodology.

In the context of this work, the following meaning is proposed for the term executable workflows. Executable workflows are integrated and executable models that are created similar to activity diagrams. They are used to model and execute structured processes for a specific task or purpose. Executable workflows consist of executable object nodes, control flow nodes and directed transitions between nodes. Executable object nodes can represent activities, processes, process steps, tasks or instances of other executable models of any domain of computation. In the case of executable object nodes that represent other model instances, these instances may represent completely different underlying models or may represent model instances of the same underlying model but with different parametrization, e.g. model instances with different seeds for random number generation.

Control flow nodes are predefined objects that are used in combination with transitions in order to determine a specific control or object flow for the overall model. Models, tasks or activities that are represented executable object nodes can be created independently from the associated executable workflow. They may exist in form of links to domain-specific models, created with different modeling languages or tools, as well as in form of integrated or external source code modules that can be executed. An executable object node may also be designed in a way to invoke an external task during workflow execution. Executable object nodes can be independent

from each other, i.e. they do not require one another for successful and complete execution, or they may interact with each other, i.e. they exchange information during workflow execution. The following general definition for executable workflows shall be used in the context of this work:

Definition 5 An *executable workflow (EW)* is an integrated model that is created to execute sets of different activities, processes, tasks or model instances in a specific order, i.e. in sequence or in parallel. More specifically, an executable workflow is a block-oriented and discrete event-based model that can be executed on a computer. It consists of hierarchically structured executable object nodes, control flow nodes and directed transitions between nodes. Executable object nodes may represent activities, processes, tasks or parametrizable and executable model instances of any computational domain and may be independent from each other or may interact in order to achieve a common goal. Model instances may relate to different underlying models or to the same underlying model while using different parameter values. Control flow nodes and transitions are used to structure and orchestrate the execution of executable object nodes.

For parallel and sequential execution of independent simulations of system models, e.g. for the optimization of system architectures, Salzwedel et al. [313], [316] and [317] proposed the development of so called simulation sets. In an early proposition for simulation sets [313], a syntax similar to activity diagrams of the Unified Modeling Language (UML) was proposed in order to connect a number of independent simulations [313].

A basic conceptual example for the graphical composition of simulation sets is shown in Figure 2.25. However, the technology to model simulation sets, e.g. for the development of automated processes during system development and optimization, is still missing. Thus, simulation sets are still a subject of current research projects, including the research project “Automatisierung der Architekturoptimierung komplexer Systeme” (ARKOSE, Engl. automation of the architecture optimization of complex systems, 2014-2017), sponsored by the *German Federal Ministry of Education and Research* under grant number 01IS13031A.

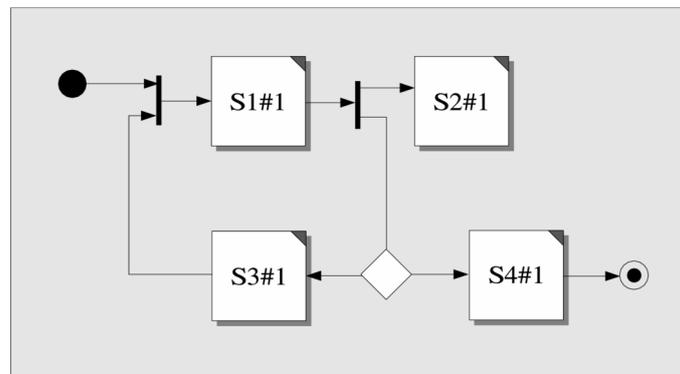


Figure 2.25 – Early concept for the graphical design of simulation sets, including control flow elements for sequential and parallel execution of simulation instances (nodes S1#1 to S4#1) (Source: Salzwedel 2007 [313])

In the context of this work, the term simulation model refers to a complete, self-contained, parametrizable model of any domain of computation that can be executed

on a computer for a specific purpose. Simulation models may exist in form of domain- and tool-specific models as well as in form of self-contained tasks or programs.

In the case of an executable workflow where each executable object node represents a simulation model instance, such an executable workflow is called *simulation set*. Thus, simulation sets represent a specific subtype of executable workflows. In this work, simulation sets are used to unite sets of simulation model instances in form of an integrated, structured and executable model. Thus, simulation sets may also be referred to as simulation of simulations. Included simulation model instances of simulation sets may represent completely different underlying simulation models or simulation model instances of the same underlying model but with different parametrization. Simulation sets may include simulation model instances that are independent from each other, i.e. they do not require one another for successful and complete execution, or they may interact with each other, i.e. they exchange information during simulation set execution. The following general definition for simulation sets shall be used in the context of this work:

Definition 6 A *simulation set* (*SimSet*) is an integrated model that is created to execute sets of coupled simulation model instances in a specific order, i.e. in sequence or in parallel. More specifically, a simulation set is a specialized executable workflow where each executable object node represents a parametrizable, self-contained and executable simulation model instance of any computational domain. Simulation model instances may represent different underlying simulation models or may share the same underlying simulation model and use different parameter values. Moreover, simulation model instances may be independent from each other or may interact in order to achieve a common goal.

2.4.3 MR-MBSE Design Environment

In order to support minimum risk model-based systems engineering (MR-MBSE), an appropriate system design and simulation environment needs to be selected that can be extended with features and predefined model components required in order to create an integrated MR-MBSE design environment. The chosen tool needs to support different levels of system abstraction for the creation of executable specifications as part of a mission level design approach. Thus, it must be possible to combine models for system behavior, architecture and environment.

For the creation of data models during executable specification development, users need to be able to determine sets of custom data structures that may include different data types, default values and value boundaries. The same applies for the development of system and component parameters that are used to describe non-functional system properties. Since complex and heterogeneous systems comprise different system domains including electrics, electronics and software, the design environment shall be capable to support different timed and untimed domains of computation. For the design of software and logical functions, e.g. finite state machines (FSM) have been proven to be successful [121]. In the case of digital systems including hardware and networks, discrete event modeling and simulation may be used [258]. Untimed simulation domain capabilities may be required to determine continuous model behavior, e.g. for analogue systems [383].

The tool shall provide a graphical modeling approach with hierarchical system decomposition to manage design complexity. With regard to the development of executable workflows and simulation sets, e.g. for automated validation and optimization, the selected tool shall provide basic capabilities to develop flowcharts, activity diagrams or statecharts.

For the development of virtual prototypes, the design environment needs to provide the means for graphical simulation visualization and animation. Moreover, it must be possible to connect external devices to the simulation model. Both features are required to control and evaluate simulations interactively. The coupling of hardware and simulation can also be used during later development stages, e.g. for hardware-in-the-loop (HIL) testing. In addition, so called plug-and-play development capabilities shall be provided. This means, that basic model components can be developed and structured within domain specific libraries and later be used to exchange entities of a model using a drag-and-drop mechanism. Plug-and-play modeling is also used for the meet-in-the-middle paradigm of the mission level design approach. Thus, abstract macro model components can be substituted with more detailed model components from later design stages. In that case, both types of component must share the same interfaces and data model.

Due to the potentially large size of complex executable specifications, the selected tool should provide the means for distributed simulations to decrease overall simulation duration. A tool with Monte-Carlo simulation capabilities is beneficial when performing a range of simulations for the extended analysis of “what if” scenarios [283]. Moreover, it must be possible to interact with external tools in order to perform more specific analyses, e.g. for data post processing. This can be achieved, for instance, by providing an application programming interface (API) or by storing models with standardized data format, e.g. by using the *Extensible Markup Language* (XML) or the *XML Metadata Interchange* (XMI) format. Complex systems like aircraft are developed by distributed teams. This requires to couple the design and validation environment with configuration or version management tools, e.g. *Apache Subversion* (SVN) or *Rational ClearCase*. Distributed teamwork also requires the generation of documentations in order to understand model components and to track changes. In terms of specification validation and system certification, capabilities for the creation of report documents are required.

Currently, a wide variety of computer-aided software and systems engineering tools exist, including in-house solutions that are created and used by large companies and organizations. Different publications provide an overview of available design tools, e.g. publications by Gartner Dataquest [334], Smith [333], Maniwa [212] and Densmore et al. [102]. In the context of this work, the system design and simulation software *MLDesigner* (cf. chapter 2.3.4.3) has been selected as an integrated platform for the development of a plug-and-play capable MR-MBSE design and validation environment. The selection was made, because *MLDesigner* already provides most features that are required in the context of this work. For example, *MLDesigner* supports different domains of computation and various levels of abstraction [102] and [3]. Moreover, it has successfully been used as part of mission level design and meet-in-the-middle system development approaches. Various examples for the successful application of *MLDesigner* for the development and optimization of different systems can be found, for instance, in references [324], [285], and [317]. Figure 2.26

depicts required MR-MBSE tool capabilities that are already supported by MLDesigner (white boxes) and necessary extension features (grey boxes) that need to be provided in the context of this work.

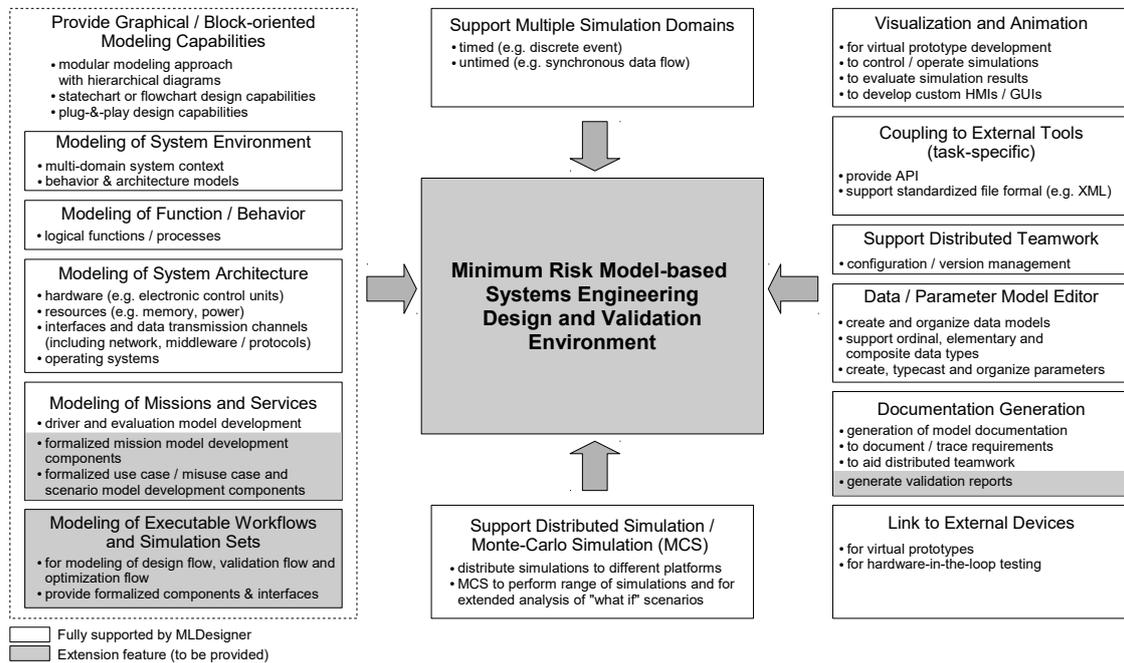


Figure 2.26 – Basic requirements, components, and capabilities for an MR-MBSE design and validation platform based on MLDesigner for the development of complex systems

2.5 Summary

In this chapter, the currently used development process for complex systems was reviewed and analyzed (cf. chapter 2.1). It was shown, that in the case of complex development projects for large systems or system-of-systems, e.g. aircraft, a high amount of product uncertainty is introduced during early design stages on the left-hand side of the V-model. Specifications with an average uncertainty of up to 70% are carried through the rest of the development process, shifting the discovery of most design flaws to implementation, test or even to system operation phase. Because expenditures to remedy specification flaws are maximized on the right-hand side of the V-model development process, the majority of complex development projects fail or experience massive overruns in development time and cost. Thus, current complex development projects only have a 6.4 % chance for full success. As a result, the currently used development approach maximizes the overall risk of development (cf. chapter 2.2). Subsequently, requirements for the early reduction of design uncertainty were elaborated together with a detailed analysis of existing solution approaches and related work, especially with regard to model-based systems engineering (MBSE) (cf. chapter 2.3). A more detailed evaluation was performed for the mission level design approach and related MBSE automation approaches.

So far, all of the described attempts have failed to deliver on their promise to reduce design uncertainty for complex aircraft systems early and thus were unable to reduce the high percentage of late specification changes and to minimize the overall risk of development [215]. The reason is that, within the used product development

life cycle of civil aircraft, top-level specifications and system specifications are not validated during early design phases to resolve design uncertainties of the coupled overall system before the beginning of subsystem decomposition and development. In the case of aeronautics software, mainly functional specifications are being created [215]. To provide both, hardware and software specifications of good quality, it is necessary to include functional and non-functional or performance requirements within the systems design and validation process [169] and [47]. This issue is still neglected during systems and software engineering [49] and [79]. The behavior of software within a real-world environment, i.e. as part of an aircraft, is a combination of hardware, software and operating systems as well as the dynamic coupling in-between [47].

In contrast to different attempts for the model-based design and validation of product specifications at subsystem level as described in chapter 2.3.3, top-level requirements specifications developed during concept design and specification are still validated by dedicated groups or individuals within a peer-review process for each part of the specification [215] and [56] (cf. existing validation methods chapter 2.2). Bounded as well as statistical uncertainties are not included in the design and validation process [313]. Moreover, the foundations for an automation of design and validation activities as part of an iterative model-based development process have not yet been implemented [316], [317] and [215].

As a result of the application of available methodologies, the overall system, i.e. aircraft, can neither be validated nor optimized early during development [215]. Thus, Marwedel et al. stated in 2010 [215], that in the case of aircraft and avionics systems: “*The level of validation required to achieve a good quality of system specifications cannot be accomplished with currently available methods [...]*” [215]. As a result, the current risk for complex system developments remains a maximum since most validation efforts are still postponed to late development stages, i.e. to the right-hand side of the V-model.

At the end of this chapter, the objectives and contributions of this work were elaborated. In this work, a model- and simulation-based systems engineering methodology is developed together with an associated design and validation environment. This is done in order to improve the quality of complex system specifications by reducing product uncertainties early at the left-hand side of the V-Model development process. Thus, the overall risk of system development shall be minimized, e.g. for the development of aircraft. In order to achieve the proposed solution, the process of validation is improved, including the provision of means for automated validation and optimization. In order to provide automation for validation and optimization processes, the use of executable workflows and simulation sets has been proposed.

3. A Model-Based Design Method for Complex Avionics Systems

This chapter presents an introduction to the proposed solution for a Minimum Risk Model-based Systems Engineering (MR-MBSE) method for complex systems with specific emphasis on the development of integrated avionics and cabin systems for large aircraft. It provides an overview of the general concept, necessary development steps and concludes with a summary.

3.1 Overview

In this work, a method for a Minimum Risk Model-based Systems Engineering approach is developed and described. The mission level design (MLD) approach was chosen as the foundation for the development of a Minimum Risk Model-based Systems Engineering approach since it already shifts design and validation efforts to early design stages at the left-hand side of the V-model, when costs for rework are still low (cf. chapter 2.3.4.2).

The general principle of the proposed MR-MBSE development process can be illustrated by complementing the top-down oriented development pyramid with an inverted validation pyramid as depicted within Figure 3.1. The inverse validation pyramid in Figure 3.1 symbolizes the amount of effort that is put into the validation of the overall design during the overall process of development. Moreover, it symbolizes that the proposed MR-MBSE method is a middle-out design approach that unites top-level mission requirements and conceptual design patterns with low level service requirements. For this purpose, the use of validated executable specifications and virtual prototypes is proposed. Both types of specification models are used in combination to shift most design decisions to early development stages and to validate and optimize the overall system design before detailed subsystem development and implementation.

Executable specifications and virtual prototypes that are developed as part of the proposed MR-MBSE method contain functional and non-functional aspects of a system under design (SuD) that are integrated within different types of sub-models, including:

- Mission / Operational models (end-user dependent)
- Quality / Performance models (end-user and system provider dependent)
- Data and customization models (end-user and system provider dependent)
- Usability models (end-user dependent)
- Functional models (system provider dependent)
- Architecture models (system provider dependent)
- System environment models (end-user dependent)

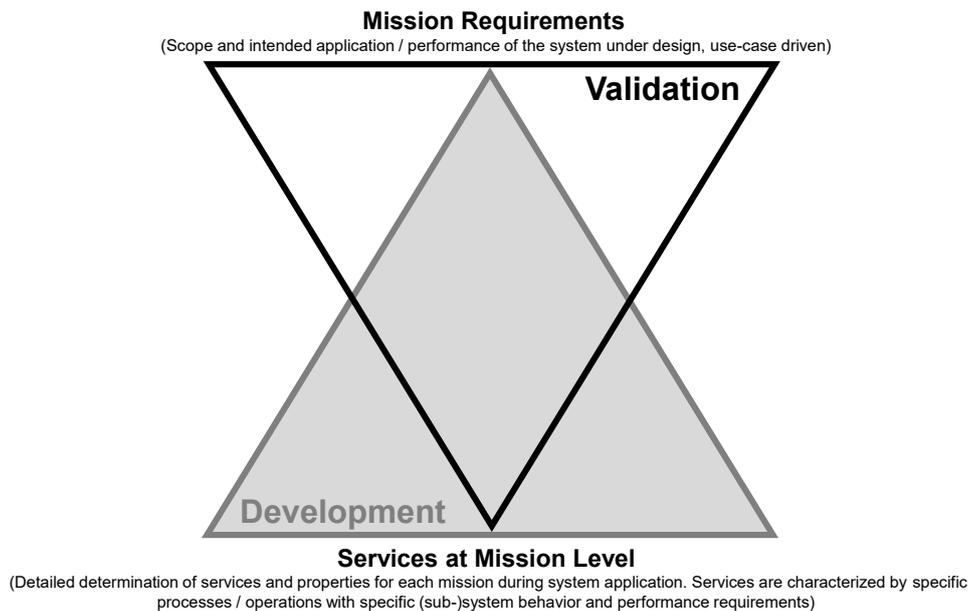


Figure 3.1 – General principle of the proposed minimum risk model-based systems engineering method: The top-down system development pyramid is complemented by an inverted validation pyramid

To improve the mission level design approach, the proposed MR-MBSE method also focuses on the development steps before mission model development and determines how to establish a consistent transition between different specification models with associated validation steps. Moreover, as an extension for the original mission level design approach, the creation of mission, use case and scenario models is formalized in this work. As part of this process, driver and evaluation models of the MLD approach are combined and the interaction between both and the overall system model is formalized in order to provide the foundation for automated validation. This includes the development of a formal system-actor-context interaction and data model and the combined validation of functional and non-functional system properties in the presence of bounded parameter uncertainties. The proposed methodology is accompanied by the development of predefined model components, e.g. for mission model development, to support users of the developed methodology from requirements specification development through to system specification development.

In order to support the development of integrated, highly configurable and customizable aircraft systems, the developed method provides means to include system

configuration and customization parameters within the proposed executable specification models. The creation of mission and service models, including scenario models, is based on executable workflows and simulation sets and follows an imperative design paradigm, i.e. designers need to determine complete and concrete sets of processes and relations between them, e.g. in order to determine a specific use case model with associated scenarios.

Figure 3.2 visualizes the proposed iterative development life cycle of MR-MBSE. At the beginning of overall development, during concept design, early concept models are created and validated that are based on the *Unified Modelling Language* (UML) or the *Systems Modeling Language* (SysML). Concept models determine stakeholder needs and top-level requirements and are used to develop a validated executable requirements specification that reflects mission and operational requirements for the SuD, including functional and non-functional requirements as well as constraints. After that, an executable overall system specification is developed that may include existing model components from previous development projects. Behavior and performance of the developed system specification are validated and optimized in two steps. Firstly, automated validation is performed by concurrent execution of both types of specification models and secondly, interactive validation is performed with stakeholder representatives during interactive simulation of the virtual prototype.

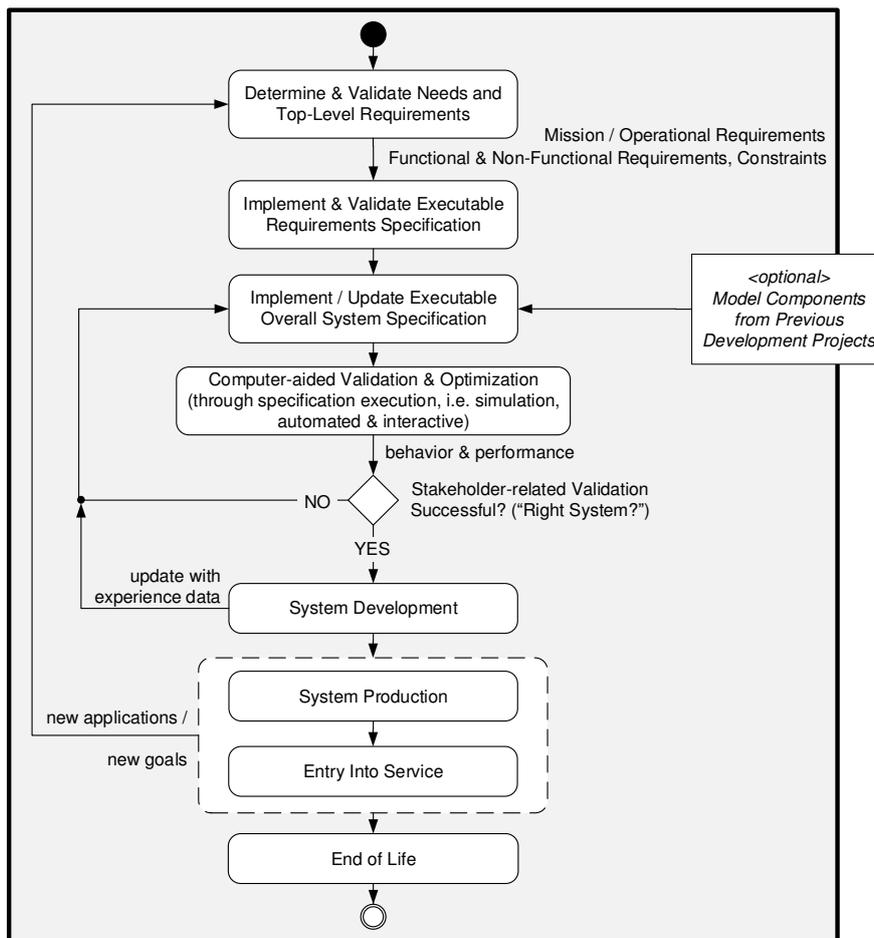


Figure 3.2 – Overview of the proposed iterative development life cycle; executable specifications used for system design and validation are applied during early design stages to increase the quality of the overall design before detailed development

During the following phase of detailed system development, experience data may be used to update and re-validate specification models. During production and use of the SuD, new goals and applications may arise that were not intended in the beginning of development. These new top-level requirements can be used to iterate the overall process in order to develop an adapted design solution. At the end of life of the SuD, validated and verified model components may be stored and used for future projects.

The proposed concept of MR-MBSE can be integrated within phase-oriented and iterative development processes such as the V-model depicted in Figure 3.3. As a result, executable specifications and virtual prototypes become the pivotal elements of the overall development process.

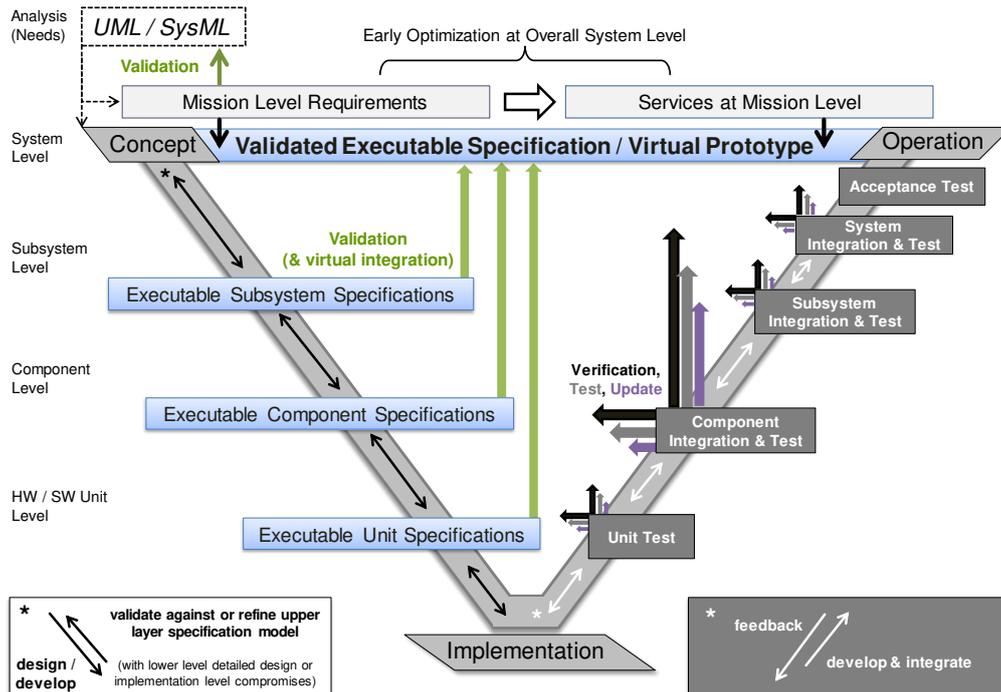


Figure 3.3 – Iterative V-model development process combined with minimum risk model-based systems engineering

3.2 MR-MBSE Concept

In this sub-chapter, the central steps of the proposed MR-MBSE method are introduced and elaborated. The central steps of MR-MBSE include requirements elicitation and concept design phase, requirements specification development, human machine interface (HMI) concept development, system specification development, overall system optimization as well as validation activities between all design stages.

3.2.1 Requirements Specification Model Development

At the beginning of concept development, Figure 3.4, diagrams of the UML and SysML are used to create concept models that are used to determine, analyze and validate top-level requirements including the determination of stakeholders, system structure, system context, intended system operations and non-functional qualities.

Based on this conceptual design, an executable requirements specification (ERS) is created. With the aid of simulation sets, an overall mission model is developed that determines one or more typical operational scenarios for the SuD. An overall mission model unites, structures and orchestrates mission requirements in the form of use cases and services, actor interaction model, data model, customization model, system context, constraints, and configuration model. This step is most important in order to ensure that the system design reflects the intended application and performance based on all stakeholder requirements.

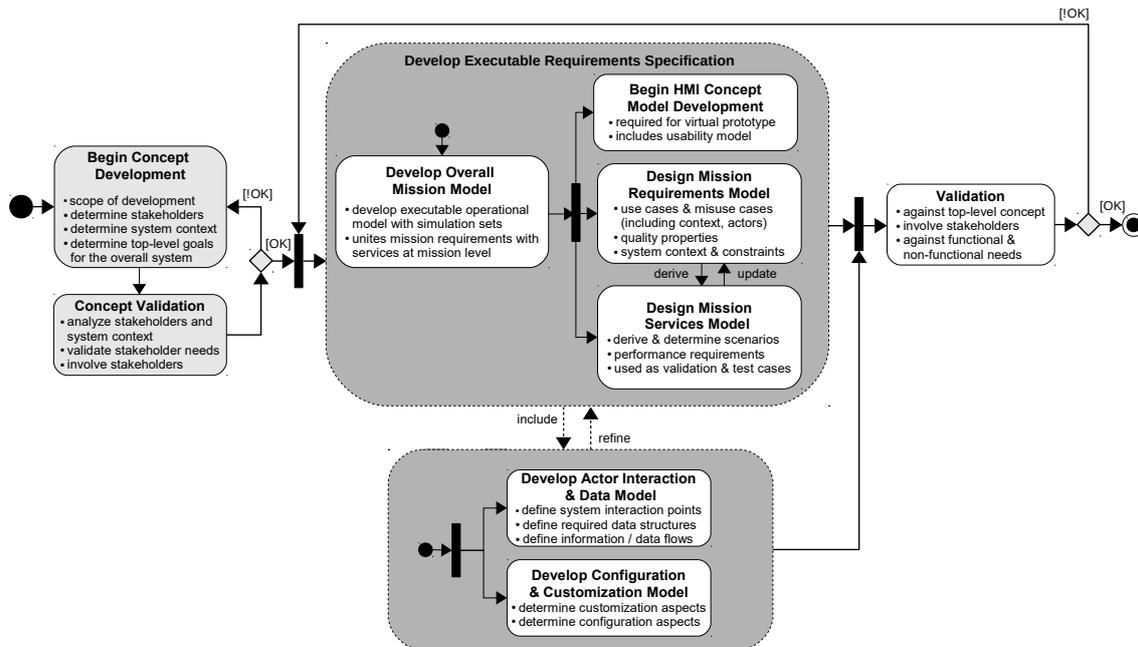


Figure 3.4 – Overview of the first phase of the proposed method – from concept design and validation to executable requirements specification and validation

Service models are used to determine concrete manifestations for each use case or misuse case. This is done by determining timed processes with preconditions, post-conditions, triggered actions, expected reactions and the related expected performance of the SuD. In addition, each scenario of the service model can be weighted, e.g. in accordance to the importance of the scenario in order to achieve the associated mission. An actor interaction model refines use case and service models by determining the specific system interfaces that relate to external as well as internal actors of the SuD for each use case. In parallel, a data model is developed that specifies structure, qualities and quantities of information that is required by the SuD during operation. It is developed in the form of sets of data structures with specific types and ranges. The data model is developed with regard to mission and scenario objectives. Since system configuration and customization aspects have a huge impact on system operation, a configuration and customization model is developed in parallel within mission and service models to determine, validate and test useful parameter permutations in the presence of a vast number of combination possibilities. It too is developed in the form of sets of data structures with specific types and ranges.

An HMI concept model is developed in parallel to the development of mission models. It is required to create an executable virtual prototype (VP) of the SuD during

later development stages, e.g. to perform coverage analyses. The HMI concept determines shape, structure, procedures and relations of interface and control elements for users of the SuD. This task includes the creation of plain audible and visual control or indication elements and the creation of comprehensive graphical user interfaces (GUI) with various menus and operation modes. The development of an HMI concept model is closely coupled with mission, use case and service model development. Thus, feedback from HMI concept development is used to update mission or service models and vice versa.

At the end of this design stage, all parts of the ERS model and HMI concept model have to be validated against the top-level requirements, i.e. the validated conceptual design. This can be done in collaboration with stakeholder representatives. After the process of validation has been completed successfully, the validated executable requirements specification (VERS) serves as the starting point for the development and validation of an executable overall system specification.

3.2.2 Overall System Specification Model Development

Following the original mission level design approach, executable overall system specification (EOSS) development starts with a top-down oriented design of an integrated model that consists of environment model (context), functional model, and architecture model as shown in Figure 3.5. The latter combines intended system structure, hardware and coupling, e.g. through networks and operation systems. Architecture models are created by following a meet-in-the-middle approach. This is done by reusing and integrating bottom-up developed and already validated model components from past development projects in conjunction with the determination of novel specification model components that are yet to be developed.

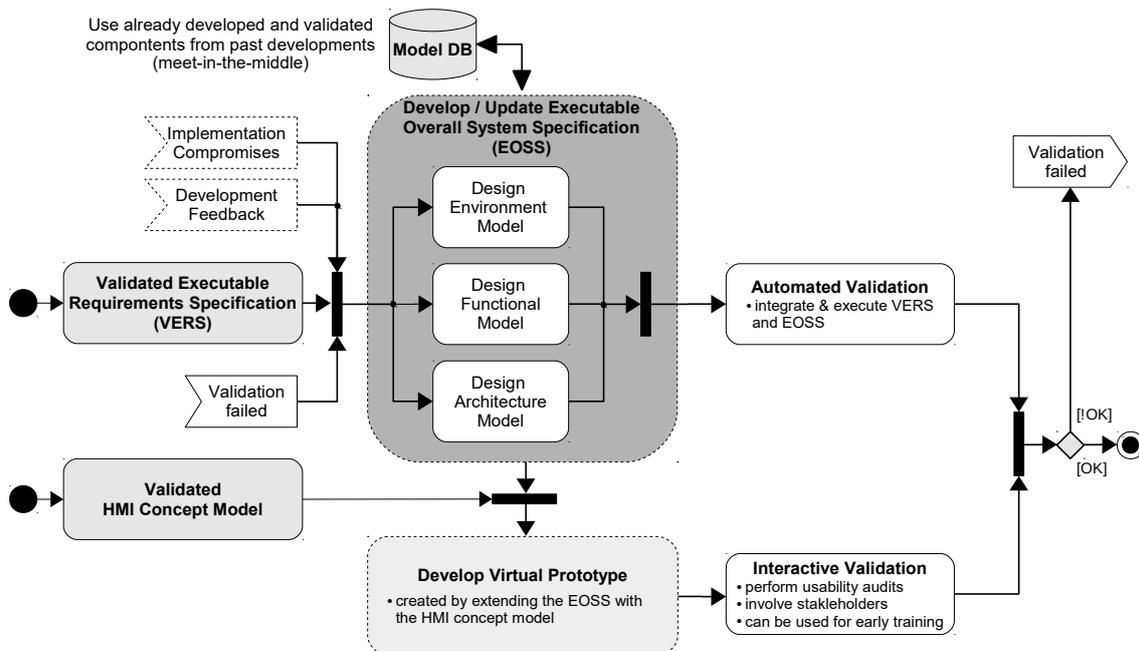


Figure 3.5 – Overview of the second phase of the proposed method – from executable overall system specification and virtual prototype development to automated and interactive validation

As part of a phase-oriented and iterative development process, an EOSS can be updated with feedback from later development stages, e.g. from detailed development or implementation. This feedback may lead to necessary design compromises that evolve from different development and implementation issues, e.g. because a non-functional quality goal is not achievable with the currently available technology. In some cases, it might even be necessary to go back even further in the development process to change the VERS or the top-level design concept in order to find a successful design compromise at overall system level.

Because of the large impact of usability and HMI aspects on the overall design of complex systems, e.g. cabin management systems (cf. chapter 1.1), MR-MBSE also provides capabilities for interactive system validation. This is achieved by creation of a virtual prototype that is based on the developed EOSS and HMI concept model in collaboration with peripheral devices (e.g. touchscreens or keyboards) as shown in Figure 3.6. During interactive validation, a VP is operated dynamically by different human stakeholders, e.g. end-user representatives. These stakeholders evaluate system functionality and performance with regard to top-level and individual requirements. Moreover, the virtual prototype can be used for early training of system operators and end users. In the case of aircraft cabin systems, this is very important because operational concepts of airline crews are highly dependent on the design and handling of available HMIs (cf. chapters 1.1 and 2.1).

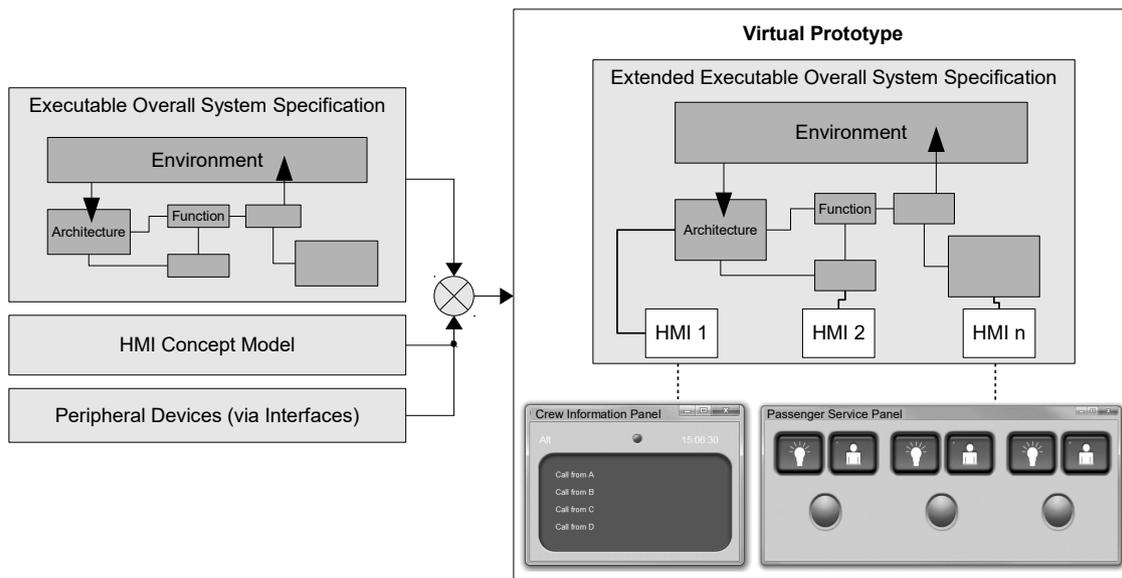


Figure 3.6 – Executable overall system specifications are extended with HMI concept models to form virtual prototypes that can be used for interactive system simulation, e.g. to perform an end-user-driven design validation or to provide early trainings for end-users.

The repeatable process of automated EOSS validation is based on the creation of a specific simulation set that is called executable validation flow (EVF). An EVF combines and orchestrates the execution of VERS and EOSS. During EVF simulation, the VERS model acts as driver and evaluation model for the SuD which is specified in the form of an EOSS. After completion of the validation process, a validation report is generated that is used to evaluate the success of the current design and to initiate the process of re-design if necessary. Figure 3.7 depicts the process from VERS development [1, 2, 3] to EOSS development [4], EVF creation,

and EVF execution [5]. During overall system simulation [5], i.e. automated validation, services models [3] interact dynamically with the EOSS model [4] by means of actors at specific interaction points [X, Y, Z]. During this process, use-case- and scenario-specific stimuli messages are created by service models and transmitted to associated actors. Thus, stimuli messages provide a cause for an expected response. Actors that relay a stimulus interact with certain interaction points of the EOSS model and may trigger sets of actions and processes within the EOSS that are observed by the same or different actors. Observed EOSS reactions are transmitted to the VERS and evaluated with regard to expected results. Information about this process is stored within a validation report.

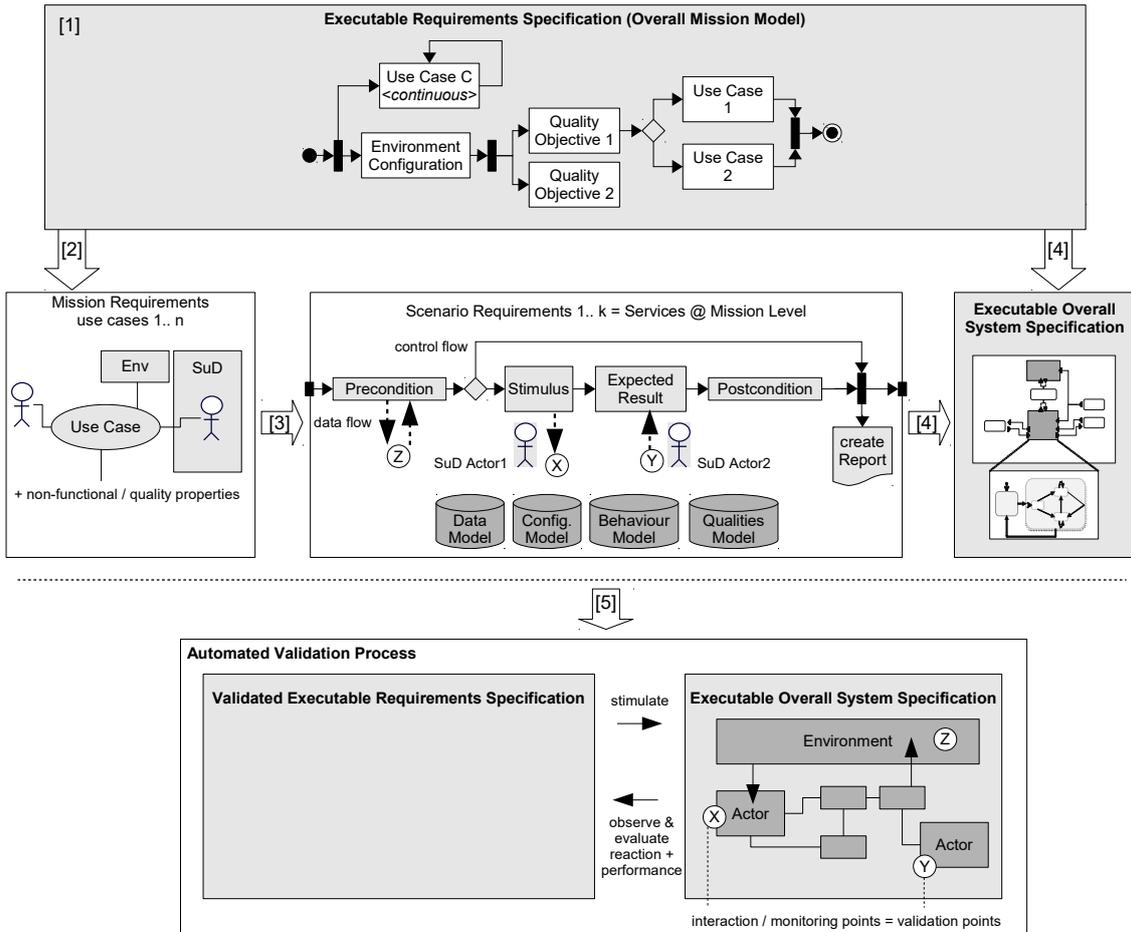


Figure 3.7 – At the beginning of executable requirements specification (ERS) development, an overall mission is developed and validated [1]. Subsequently, mission requirement models [2] with associated scenario models, i.e. services at mission level [3], are created and validated. Based on the validated ERS [4], an executable overall system specification (EOSS) is developed. Finally, an executable validation flow is developed that combines ERS and EOSS models to execute an automated validation process [5].

In contrast to other available model-based validation automation approaches (cf. section 2.3.5), it is the aim of MR-MBSE to automate the process of validation by using VERS in a way to evaluate if the system design that has been specified within an EOSS satisfies all stakeholder requirements rather than to verify or test a specific implementation. Hence, mission and use case models view the overall system as a black box or gray box that needs to provide a set of services with related performances between different actors or, in a similar manner, needs to suppress

misuse. This is because the process of validation does not need to have a profound knowledge of how specific functions are designed, structured or implemented. Automated validation ensures the design of “the right system” by evaluating that the intended application or mission for the SuD can be achieved with adequate overall performance (cf. chapter 2.3.1).

After the overall process of validation has been completed, the validated EOSS (VEOSS) permits to perform an early optimization of the integrated overall system with regard to the top-level requirements of the SuD as depicted in Figure 3.8. With the aid of simulation set capabilities, an executable optimization flow (EOF) can be designed that unites optimization model, evaluation model (with objective function) and EOSS. In that case, a finite number of consecutive steps is executed automatically in order to find the best possible solution to a given design problem. This process may include several iterations of architecture model generation, behavior and performance validation and evaluation. After the design has been optimized at overall system level, the specification is distributed to subsystem developers in order to begin the process of detailed development with subsystem specifications and component specifications.

Since MR-MBSE is phase-oriented and iterative, each subsystem may use higher level requirements specifications in order to determine more detailed subsystem mission and service models. Based on the specific knowledge of subsystem specialists, detailed formal product specification models are developed that include domain specific knowledge of the particular subsystem. By using value ranges for performance requirements and parameters, the overall design can be validated in the presence of bounded uncertainties. Detailed specifications at subsystem, component or unit level are used for bottom-up virtual system integration according to the mission level design approach. By this it is possible to validate and verify if results from detailed development meet requirements specifications in terms of functional and quality properties.

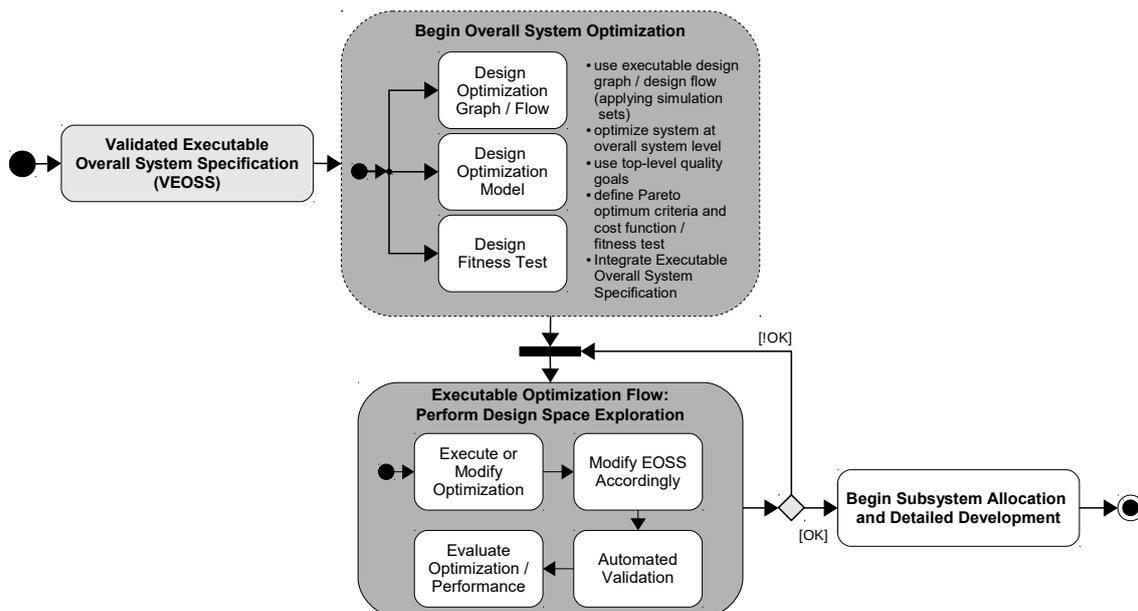


Figure 3.8 – Generic flowchart for performing overall system optimization with executable optimization flows before detailed subsystem development

The use of formal executable specifications as part of an MR-MBSE development process also enables the application of additional methods to improve design quality by means of formal specification model analysis and verification. For instance, in case system functions are described with automata, e.g. finite state machines, completeness and consistency can be mathematically verified [380]. During implementation, executable product specifications that have been created in response to executable system specifications at system, subsystem, component, or unit level can be used for verification. See reference [254] for more information on automated verification of executable specifications. With regard to the different model-based test approaches described in chapter 2.3.5, VERS and EOSS can be reused for testing during later development stages.

3.3 Summary

This work deals with the early reduction of product uncertainty within specifications to minimize the overall risk of development, i.e. the risk that complex development projects will fail or experience massive delays and cost overruns. In order to achieve this goal, a minimum risk model-based systems engineering methodology is developed in this work that uses executable specifications and virtual prototypes in collaboration with automated and interactive validation processes. In this chapter, the concept for the proposed methodology was introduced. Moreover, major steps of the proposed method were elaborated, including requirements specification model development, overall system specification model development, automated and interactive design validation as well as automated design optimization.

4. Minimum Risk Model-based Systems Engineering Method

In this chapter, the minimum risk model-based engineering (MR-MBSE) approach proposed in chapter 3 is developed in detail. Firstly, fundamentals for the development of executable workflows and simulation sets for MR-MBSE are determined more closely together with corresponding model components. Executable workflows and simulation sets are applied for a wide variety of applications during MR-MBSE, including the automation of validation and optimization activities. Secondly, a requirements elicitation and analysis process is elaborated for the development and validation of top-level concept designs. This process utilizes a use case and scenario-based requirements analysis, specification and validation approach. Based on the validated conceptual design, executable requirements specifications (ERS) are developed. ERS combine mission and service models for the system under design and include early human machine interface concept models. After ERS validation, the concepts of executable system specifications and virtual prototypes are elaborated. Following the description of the overall system specification development process, automated as well as interactive validation strategies are described together with the proposed process of design optimization automation. Each sub-chapter includes a detailed description of plug-and-play capable modeling components that have been developed in parallel the proposed methodology (cf. appendix A).

4.1 Executable Workflows for MR-MBSE

In chapter 2.3.5.5 and chapter 2.4.2 it is described how executable workflows are used in economics or science to structure executable processes in order to create repeatable process structures. The same basic principle is used within this work in order to develop executable workflows adapted for MR-MBSE as well as simulation sets to enable the development of executable requirements specifications and to automate the process of validation and architecture optimization. Moreover, both types of executable workflows can be used for other iterative processes, e.g. to evaluate changes within system models or to validate refinements of a specification model resulting from more detailed levels of systems design during later stages of

development. In this sub-chapter, elementary modeling concepts are described for executable workflows. After the description of general workflow elements and structures, the specific application of executable workflows in terms of simulation sets is elaborated in section 4.1.3.

Executable workflows consist of executable object nodes, control nodes as well as transitions (edges) which form directed graphs for describing finite control flows. The information flow between elements of an executable workflow can be modeled by accessing external databases, files or global data storage modules (also referred to as memories). Executable object nodes may represent instances of activities, missions, simulation models, composite models, source code primitives or statecharts. An executable workflow has one or more initial nodes and one final node. Executable workflows (EW) can be described by a 6-tuple $(O, C, D, P_{EW}, I, c_n)$ where:

- $O = \{o_1, o_2, \dots, o_i\}$ is a finite non-empty set of executable object nodes
- $C = \{c_1, c_2, \dots, c_n\}$ is a finite non-empty set of control nodes
- $D =$ is a finite set of shared memories / data storage modules ($\emptyset \in D$)
- $P_{EW} =$ is a finite non-empty set of configurable parameters
- $I \subset C =$ a finite non-empty set of initial control nodes
- $c_n \in C$ is the final node

In the context of MR-MBSE, executable object nodes may only be chosen from a finite set of node types: $\forall o \in O: o \in \{MIS, SIM, MOD, PRM, FSM\}$ where:

- MIS is a finite set of mission models and service models
- SIM is a finite set of simulation models
- MOD is a finite set of hierarchical composite block models
- PRM is a finite set of primitives, i.e. activities, processes, tasks or models that are described by executable source code
- FSM is a finite set of finite state machines

Simulation models, composite models and primitives may be modeled in any domain of computation. Mission and service models (MIS) are modeled using the discrete event domain (top-level) but may also include other domains at lower levels of hierarchy. Each mission contains a set of atomic missions, services and scenarios. A detailed description of missions and scenarios is provided in section 4.3.1 and section 4.3.5. Simulation models represent the highest level of model abstraction. The use of composite models (MOD), primitives (PRM) and finite state machines (FSM) is described in section 2.3.4.3. Since composite models can nest any type of computational domain or sub-model, it is also possible to define new types of executable object nodes if necessary. More detailed information on discrete event simulation can be found e.g. in references [27] and [386].

Figure 4.1 depicts an exemplary structure for an executable workflow. This structure consists of different executable object nodes and control nodes that are connected through directed transition edges. The control flow of an executable workflow describes the order in which workflow elements are executed and is driven by tokens that proceed from node to node, beginning at initial control nodes. When the control flow arrives at the final node, the simulation of the executable workflow is finished. Executable object nodes have exactly one input and one output port. They become active by receiving a token at their input port and finish execution by releasing a token at the output port. Control nodes also receive and release tokens according to their defined purpose. For instance, control nodes can be used to split the control flow or join different branches. The different types of control nodes will be elaborated more closely later in section 4.1.1.

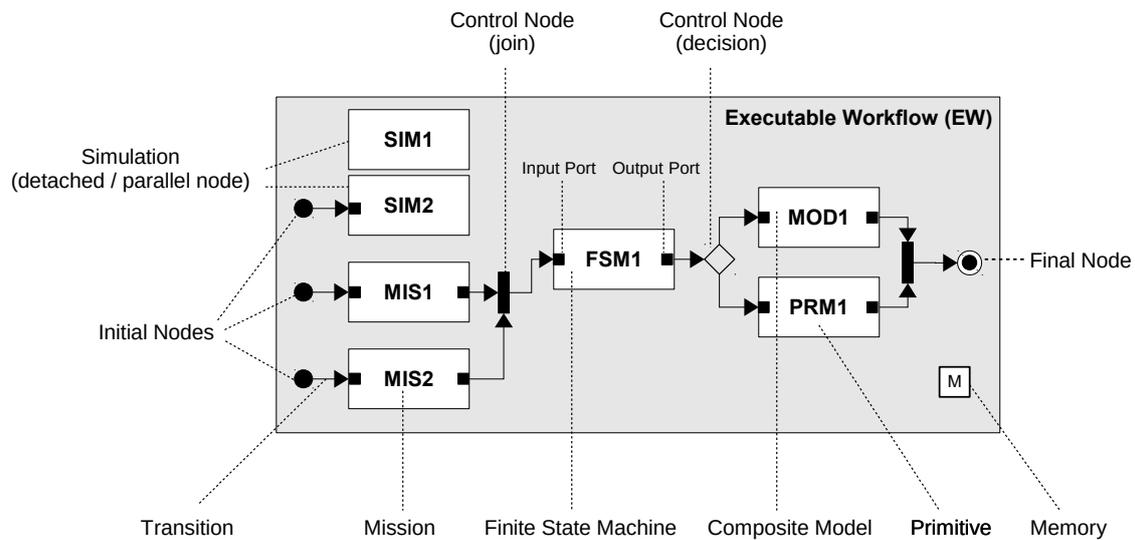


Figure 4.1 – Example of an executable workflow with different executable object nodes and control nodes

Executable workflows also permit use executable object nodes with no connections to the control flow graph. These nodes are called parallel or detached nodes. In this case, the executable object node has no input or output ports and starts executing when the overall workflow is executed. Another form of parallel node is equipped with an input port that is used to trigger execution at a specific point during workflow execution. Execution of both node types finishes when the overall workflow execution has ended by reaching the final control node. Executable workflows should include at least one non-detached node. In the upper left corner of Figure 4.1, two parallel executable object nodes of type simulation are depicted. On the lower right, a memory module is depicted. Memory modules do not have connections to any node and are directly accessed by executable object nodes to exchange information. More information on data exchange within executable workflows is provided in section 4.1.2.

An executable workflow also contains default parameters as well as a finite set of user-defined parameters. Default parameters include a global seed for random number generation functions and a value to define a fixed or infinite run length for workflow execution. Custom parameters can be defined and linked to executable object nodes

or memories or directly exported from already defined parameters of executable object nodes, e.g. in order to manage node configurations at workflow level. By configuring a finite number (2..n) of so-called parameter sets (PS), it is possible to automatically execute different consecutive iterations of the same workflow with different parameter configurations. A parameter set PS_k represents a configuration of specific values for each parameter $p \in P_{EW}$. For each parameter set, the workflow is executed once.

Note: A possible extension for executable workflows is to extend the set of possible types for executable object nodes by incorporating lower level executable workflows so that: $\forall o \in O: o \in \{EW_{L-1}, MIS, SIM, MOD, PRM, FSM\}$ and EW_{L-1} is a finite set of executable workflows at a lower hierarchical level. Any executable workflow at a higher hierarchical level $ew_x \in EW_L$ must not be part of any workflow at a lower hierarchical level: $ew_y \in EW_{L-i}: EW_L \cap EW_{L-i} = \emptyset, EW_L \cup EW_{L-i} = EW$. A similar extension is applied for the development of mission models in chapter 4.3.1.

4.1.1 Control Flow and Control Nodes

As described before, the control flow of executable workflows is driven by tokens that pass along transition edges between executable object nodes (O) and control nodes (C). Tokens can be produced and consumed by both types of nodes.

Definition 7 A *token* is an event object, i.e. data particle, of executable workflows that gives priority to every node receiving it at its input port. Any executable workflow node receiving a token becomes active and is able to execute. Tokens can be created, evaluated and consumed.

Tokens are created, evaluated and consumed by different control nodes or executable object nodes. They are exchanged between input and output ports of executable object nodes and control nodes in the form of an integer type data particle t with $t \in \{0,1,2\}$. The specific value of a token is only used by decision-based control nodes. If an executable object node receives a token, it is executed immediately. As soon as its execution is finished, a token is produced at the output port of the node in order to hand over control to the next node in line. At this point it is important to note, that the designer of any type of executable object node is required to secure, that the specific node terminates eventually in order to keep the executable workflow alive. This can be done, for instance, by providing a time-out condition for every executable object node. Otherwise, the execution of a workflow may not terminate. In the context of this work, an executable object node cannot be re-triggered while executing. In this case, any token received will be discarded. Transition edges may exist between any type of node within an executable workflow. Thus, transition edges for defining a control flow have the following relation F : $F \subseteq (O \times O) \cup (O \times C) \cup (C \times O) \cup (C \times C)$

Any executable object node that is arranged as parallel node within an executable workflow has no input or output ports and is executed immediately at the begin of the overall workflow execution. Parallel or detached executable object nodes can be compared to continuous simulation model components that are active as long as an executable workflow is executed. Thus, a parallel executable object node ends

execution only in case the execution of the overall workflow has ended. If any other executable object node of an executable workflow is dependent in any way on a parallel executable object node, the designer of the parallel executable object node needs to ensure, that the respective node does not terminate before the end of overall workflow execution. Although possible, executable workflows should not consist of parallel executable object nodes only.

The control flow of any executable workflow begins at one or more initial control nodes and ends with one final node as depicted in the example executable workflow in Figure 4.2. Active nodes and transitions are colored in blue. At simulation start-up t_0 , the parallel executable object node *SIM3* becomes active and starts executing immediately. At the same time, active control is handed over from the initial control node to executable object node *SIM1* via transition $e1$. *SIM1* executes for a finite amount of time t_{S1} . When finished ($t_1 = t_0 + t_{S1}$), *SIM1* hands over active control to node *SIM2* by sending a token via transition $e2$. Thus, *SIM2* starts executing for a finite amount of time t_{S2} . At the time *SIM2* ends execution ($t_2 = t_1 + t_{S2}$), a token is sent via transition $e3$ to the final control node of the executable workflow. At the same time, the execution of parallel node *SIM3* ends.

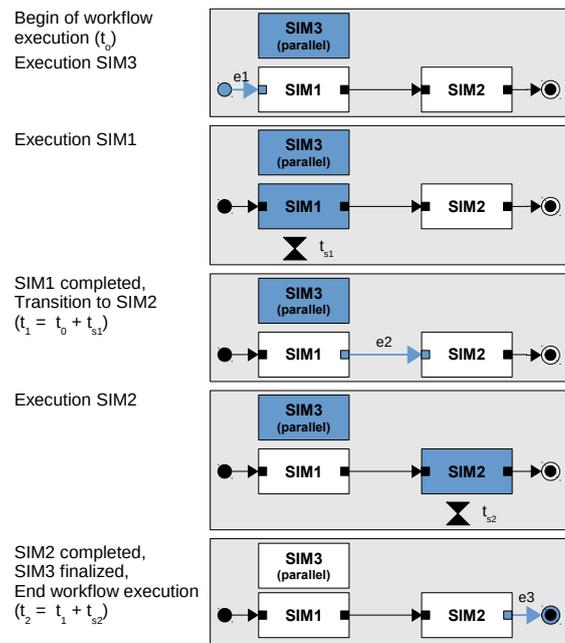


Figure 4.2 – Token-based control flow sequence with two executable object nodes, one parallel executable object node, one initial control node and the final control node

In general, a control node is used to shape the structure of any executable workflow starting from a set of defined starting nodes and ending at one final node. In other words, control flow nodes provide users with the ability to execute executable object nodes in any order. In the following section, control nodes that have been created in the context of this work are elaborated more closely. As part of this work, a basic set of predefined control nodes has been created for the design of executable workflows with regard to the tasks of automated design validation and optimization. However, the set of possible control nodes is not limited and can be extended. Control nodes are also not limited to elements of a specific diagram type, e.g. activity diagrams of the Unified Modeling Language (UML). Users may develop

different control nodes according to the needs of a specific design problem. In that case, custom control nodes shall only be designed in a way to support the token-based control flow described earlier. As an example, a timed initial control node could be determined that starts a token-based control flow after a specific amount of simulation time. Another type of node could be used in order to finish the overall execution of a workflow after a certain amount of simulation time.

Figure 4.3 shows the initial control node (left), the final control node (middle) and the branch end control node (right). Initial control nodes have only one output port and produce exactly one output token at simulation start-up. They are used to determine specific starting points for any executable workflow. In contrast, a final control node has only one input port and is only used once within an executable workflow. Once it receives a control token at its input port, the execution of the overall workflow is ended. This includes the finalization of all parallel executable object nodes. A branch end control node may be used in order to end a specific sub-branch of an executable workflow. When this node receives a token, the token is discarded with no effect on the overall workflow execution. A branch end control node can be connected to an executable object node.

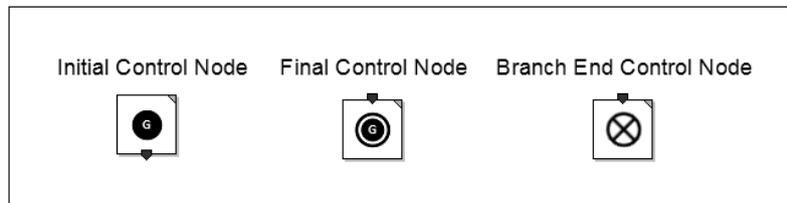


Figure 4.3 – Initial control node (left), final control node (middle) and branch end control node (right)

In order to split up a control flow, a split control node is available. With respect to the graphical modeling approach, two different split control node shapes are available as shown in Figure 4.4. Both node types are designed with a dynamically adjustable number of output ports to provide users with the ability to select an arbitrary number of output ports (2..n) during modeling. This is done via parameter configuration, e.g. to select two or four synchronously firing output ports as depicted in Figure 4.4. The split control node is designed to fork the control flow synchronously. This means, that upon receiving a token at the input port, the token is multiplied for each output port and sent at the same time (synchronous event).

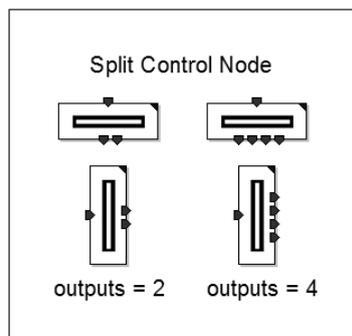


Figure 4.4 – Split control node; Horizontal (upper pair) and vertical (lower pair) node shapes with configurable number of output ports are available.

In contrast to the split control node, the synchronized join control node depicted in Figure 4.5 is used to merge multiple input branches of the control flow into one output transition. Only if at least one token has been received at each input port, the output port is activated once in order to produce one token.

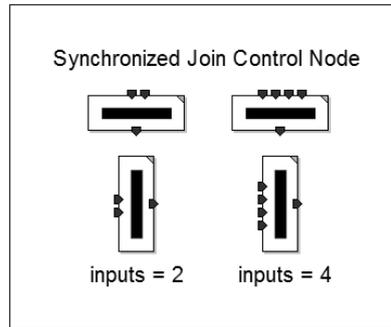


Figure 4.5 – Synchronized join control node; Horizontal (upper pair) and vertical (lower pair) node shapes with configurable number of input ports are available.

After a token has been produced, a new token may only be produced again if all input ports have received at least another token. Thus, this control node is used to synchronize different branches of the control flow. Two different node shapes are available for graphical modeling, one horizontal and one vertical shape. Moreover, the synchronous join control node can be adjusted in order to provide a configurable number of input ports.

In Figure 4.6, the unsynchronized join control node is depicted (multiplexer). It is also used to merge multiple input branches of the control flow into one output transition. Other than the synchronized join control node, it produces an output token each time any of the input ports has received a token. In case several or all input ports receive a token at the same time (equal time stamp), only one token is sent via the output port. Thus, no more than one control token can be sent during any simulation time stamp. As shown in Figure 4.6, this control node can be configured to have an arbitrary number of input ports that can be arranged graphically as required.

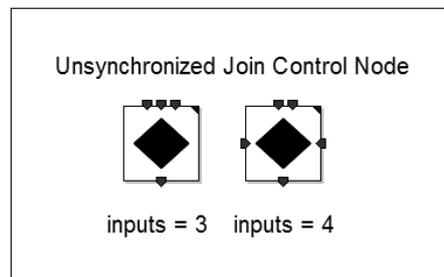


Figure 4.6 – Unsynchronized join control node with configurable number of input ports are available, e.g. 3 input ports (left) or 4 input ports (right)

To provide modelers with the ability to split the control flow based on conditions that are related to the execution of a preceding executable object node, two types of decision control nodes have been created as shown in Figure 4.7.

From the set of control nodes described in this section, decision control nodes are the only ones that evaluate the numerical value of a control token. Thus, a control

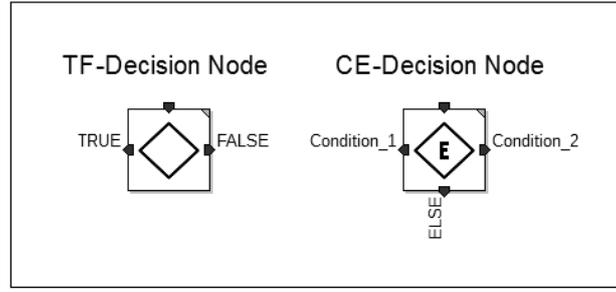


Figure 4.7 – TF-decision control node (left) and CE-decision control node

token does provide a limited amount of information to any decision control node. The TF-decision control node depicted on the left-hand side of Figure 4.7 has one input port I and two output ports called O_{TRUE} and O_{FALSE} . Moreover, this node has a configurable text parameter field that is used to state a condition related to the decision control node. Incoming token values are evaluated in order to create an output token at the respective output port:

$$O_{TRUE} = \begin{cases} (\text{new})\text{Token}, & \text{if Token.value} \in I = 1 \\ \emptyset, & \text{else} \end{cases}$$

$$O_{FALSE} = \begin{cases} \emptyset, & \text{if Token.value} \in I = 1 \\ (\text{new})\text{Token}, & \text{else} \end{cases}$$

The CE-decision control node depicted on the right-hand side of Figure 4.7 can be used to split the control flow based on a set of conditions. In contrast to the TF-decision control node, this decision control node type has one input port I and three output ports called $O_{Condition_1}$, $O_{Condition_2}$ and O_{ELSE} . The first two output ports $O_{Condition_1}$ and $O_{Condition_2}$ are related to specific condition while the last output port O_{ELSE} represents an alternative output port that is triggered when no other condition is evaluated to be true. This node has two configurable text parameter fields that are used to state conditions related to the decision control node. Incoming token values are evaluated in order to create an output token at the respective output port:

$$O_{Condition_1} = \begin{cases} (\text{new})\text{Token}, & \text{if Token.value} \in I = 0 \\ \emptyset, & \text{else} \end{cases}$$

$$O_{Condition_2} = \begin{cases} (\text{new})\text{Token}, & \text{if Token.value} \in I = 1 \\ \emptyset, & \text{else} \end{cases}$$

$$O_{ELSE} = \begin{cases} \emptyset, & \text{if Token.value} \in I = 0 \vee 1 \\ (\text{new})\text{Token}, & \text{else} \end{cases}$$

In order to split a control flow according to a specific condition, a preceding executable object node, e.g. of type composite model, is required in order to prepare and evaluate the decision in detail and to provide an output token with specific value, e.g. $v \in \{0,1,2\}$. This is because control nodes cannot access any information related to executable object nodes. As an example, an executable object node may

evaluate a set of system parameters that were changed during the execution of an object node of type simulation. If an optimum set of parameter values was found or an abort criterion has been fulfilled, a token with value 1 is created that will trigger the O_{TRUE} port of a succeeding TF-decision control node. Otherwise, a token with value 0 or 2 may be created that will trigger the O_{FALSE} port of a succeeding TF-decision control node in order to iterate the overall process by repeating the executing of a set of preceding executable object nodes.

Timed event control nodes and random timed event control nodes can be used in order to model workflows branches or executable object nodes that are contingent on the passing of a certain amount of simulation time. For instance, these control nodes may be used in combination with executable object nodes that need to be triggered or re-triggered after a specific amount of time.

Figure 4.8 shows both types of timed control nodes. Both node types gain active control via receiving a control token at the input port. Timed event control nodes provide two parameters. Parameter “*Event_Trigger_Interval*” is used to determine a specific amount of simulation time. After becoming active, the specified amount of time needs to pass before a control token is generated at the output port, thus triggering an attached executable object or another control node. Parameter “*Continuous*” can be set true or false. In case the parameter was set to false, only one output event is created. If set to true, the respective timed event control node will continue to output control tokens in regular intervals as determined by the first parameter.

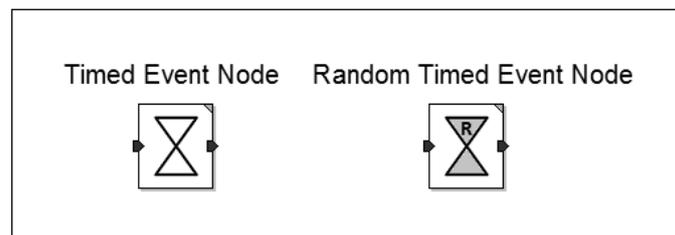


Figure 4.8 – Timed event control node (left) and random timed event control node (right)

Random timed event control nodes operate similar to timed event control nodes. After becoming active, a random amount of simulation time needs to pass before a control token is created at the output port. Three different parameters exist in order to adjust randomization. Parameter “*Distribution*” is used to specify the distribution used by the random number generator. Five different distributions can be chosen: Binomial, exponential, normal, Poisson and uniform distribution. Parameter “*MaxOrVarianceOrProbability*” specifies the variance for normal distribution, the maximum value for uniform distribution, or the probability of a single trial for binomial distribution. It is not used for exponential and Poisson distribution. The third parameter is “*MinOrMeanOrTrials*”. This parameter specifies the mean for exponential, normal, and Poisson distribution, the minimum value for uniform distribution, or the number of trials for binomial distribution. The last parameter is “*Continuous*”. Again, this parameter can be set true or false and is used to output either a single control token or a sequence of tokens with random amounts of simulation time between each token.

4.1.2 Information Exchange and Data Flow

In order to enable the development of executable workflows that are able to execute a number of successive simulation models that interact with each other, it is imperative to provide the means for information exchange. This is done by providing several data flow mechanisms for executable object nodes. As indicated in section 4.1, three general ways exist in order to exchange information between executable object nodes. As depicted in Figure 4.9, information can be exchanged unidirectionally and bidirectionally (blue lines) between executable object nodes by accessing external databases and files (blue) or internal memory modules during workflow execution.

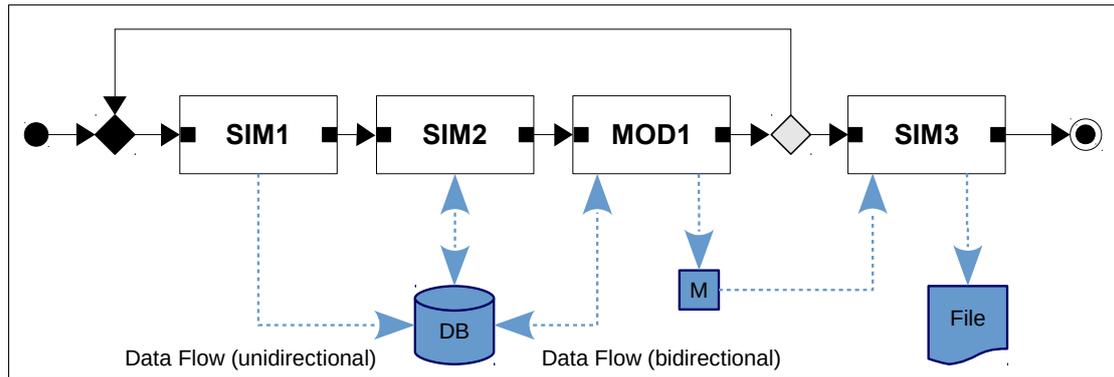


Figure 4.9 – Executable workflow example with highlighted data flow (blue) between executable object nodes using an external database (DB), a file and a memory module (M)

When using external databases during workflow execution, it is important that all executable object nodes use the same data model underlying each database. In general, using the same data model is important for all types of information exchange described within this section.

The second possibility to exchange information between executable object nodes is to use non-binary files. Moreover, non-binary files are an important tool for post-simulation analysis, e.g. by using a file generated during workflow execution for documentation purposes or as input for other software tools. It is possible to define any non-binary file types including Extensible Markup Language (XML), JavaScript Object Notation (JSON) or Character-Separated Values (CSV) files. It is possible to define files to be used for each executable object node with parameters specifying file name and type at workflow level. In case files used by several executable object nodes are created during workflow execution, it is important to ensure that the file has been created and linked correctly in order to avoid possible side effects or deadlocks.

Memory modules are internal parts of executable workflows. They are shared model elements with unique identification and are used to exchange information between different executable object nodes. For that purpose, any executable object node may link to any memory. Moreover, memories can be linked to model components at any hierarchical level of an executable object node. Memories usually point to specific locations within the physical memory of the simulation platform. In other words, memory modules can be compared to variables that store information in the form of complex data types with defined ranges and initial values. A memory can be configured to store basic data types, arrays, enumerations or composite types.

Memories can be initialized directly during workflow execution start-up. For this purpose, a set of predefined parameter fields is available. Moreover, memories can be typed and initialized by any executable object node. In that case, the respective memory does not need to be configured before workflow execution.

In addition to the three information exchange strategies described, other options may be added in order to exchange information between nodes, e.g. for dynamical information exchange between executable object nodes and parallel executable object nodes. A task-specific way for information exchange in the context of mission models is elaborated more closely in chapter 4.3.1 and used later in chapter 4.4.2.

4.1.3 Towards Simulation Sets

In the case of an executable workflow where each executable object node represents a self-contained simulation model or a complete and self-contained external task, such an executable workflow is called *simulation set* (cf. Definition 6, chapter 2.4.2). Thus, a simulation set may only contain executable object nodes of type simulation model (SIM) as well as control nodes. Simulation model nodes can be modeled with domain-specific tools in any domain of computation or by developing a self-contained executable program. The latter may be necessary in order to integrate simulation models in the form of external processes and tasks.

In most modeling and simulation tools, it is possible to run simulations independent from the graphical user interface (GUI). In order to do so, simulation models that have been created in any domain need to be transformed into a so-called external compilation. An external compilation contains all files necessary to simulate a specific system model. External simulations can be created e.g. in the form of C++ or Ptolemy Tcl interpreter code (PTcl) which is based on the tool command language (Tcl). Both types of code can be executed by using the respective command environment.

A second way to execute external compilations is to use dedicated process handling primitives. For example, different MLDesigner source code primitives have been developed by Jungebloud (cf. reference [368]) that enable the execution of external tasks and programs during simulation. Figure 4.10 depicts three primitives that were developed by Jungebloud in order to handle external process calls during simulation.

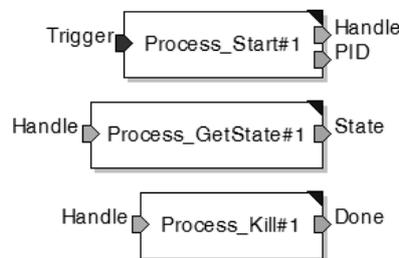


Figure 4.10 – Examples for external task and process handling primitives

Source code primitives of type *Process_Start* are used in order to trigger the start of a process during simulation. The primitive has parameter fields in order to specify an executable external process or program together with additional arguments. For

each triggered process, a unique process id and handle are provided at the output ports of the primitive. This handle can be stored and used to access the current state of a process (primitive *Process_GetState*) or to kill it at any time during simulation (primitive *Process_Kill*).

Using process handling primitives allows to execute an arbitrary number of simulation model instances during system simulation. In order to do so, simulation models developed in the form of hierarchical block diagrams need to be generated in the form of external source code. This code can be executed, e.g. with a command shell. By configuring a *Process_Start* primitive to start a new command shell process that uses a previously generated PTcl script, an instance of the respective simulation can be executed, monitored or ended dynamically during simulation. The same primitive can also be used to execute simulations with external tools. Both types of simulation models are called external simulations in the context of this work.

To be able to exchange information between different simulation models, these models need to agree on a common data model and data flow. In the case of using external simulation model compilations or external simulation models designed with domain-specific tools, information should rather be exchanged by using databases or non-binary files instead of executable workflow memories. By this, it is not necessary to provide a tool-specific interface between the executable workflow and any other simulation tool since information from any simulation model instance can be accessed independently. By using the developed process handling primitives, a dedicated executable object node has been developed that can be used for the development of simulation sets. This node controls the execution of an external simulation model instance. Figure 4.11 shows an example simulation model that has been created for executing external simulation instances as part of an associated simulation set.

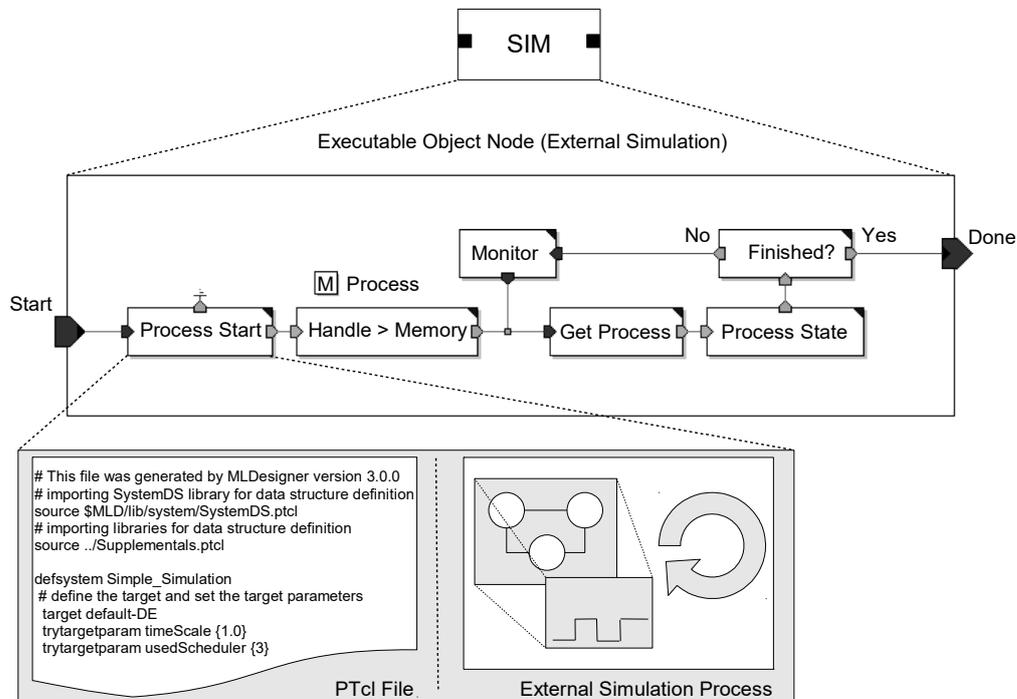


Figure 4.11 – Model for executing and handling external simulations as part of a simulation set

The module depicted in Figure 4.11 contains four parameter fields. Parameter “Program” is used to determine an executable external process (program file). This process is called with optional arguments specified in parameter “Arguments”. The two remaining parameters are “Timeout” and “Polling_Interval”.

After becoming active by receiving a control flow token, an external simulation process is started. If the attempt to start the specified process fails, execution of the module is finished and a token is created at the output port. After successfully starting an external process, the state of the process is monitored. The parameter “Polling_Interval” can be used to define a polling frequency.

It is also possible to force the end of an execution of an external process by defining a positive number for the parameter “Timeout”. If set to -1, no timeout is used. In that case, it is essential for the external process to terminate after a finite amount of time. Otherwise a deadlock is created and the simulation set will not end properly. An overview of developed executable workflow and simulation set model components is provided in appendix A.2.

4.2 Conceptual Design with MR-MBSE

As described in chapter 2.1, the process of systems engineering is well documented. A detailed introduction and description for the design and development of systems beginning with conceptual and preliminary design has been provided by Blanchard and Fabrycky [39]. Pohl and Rupp provided a comprehensive guide to the fundamentals related to systems and requirements engineering [268] while Weilkiens defined an approach called Systems Modeling Process (SYSMOD) that describes the process of requirements engineering for early design phases based on the Systems Modeling Language (SysML) and Unified Modeling Language (UML) [365]. For more information on general aspects of concept design, please refer to the references listed above.

In this chapter, the early stages of the proposed minimum risk model-based engineering (MR-MBSE) process are described, beginning with conceptual design. This is necessary in order to define a consistent design flow, leading from concept design to automated validation.

Figure 4.12 depicts an activity diagram for the process of concept design and validation. Each activity of the process (ovals) generates an output (rectangles on the left), e.g. in the form of a model. The process commences with the definition of a central goal of development, leads to concept development and ends with the initiation of the process of executable requirements specification development. In the following sub-chapters, each step is elaborated more closely.

The process depicted in Figure 4.12 is considered part of an iterative and phase-oriented development process, e.g. the V-Model described in chapter 2.1. Thus, parts of the process are iterated on and the sequence of execution may shift in the course of development. A good example is the process of determining system context and stakeholders. Both processes are mutually dependent. During requirements elicitation and analysis, new system context information and stakeholders may be identified that are important for the system under design (SuD) in order to provide certain service and quality properties.

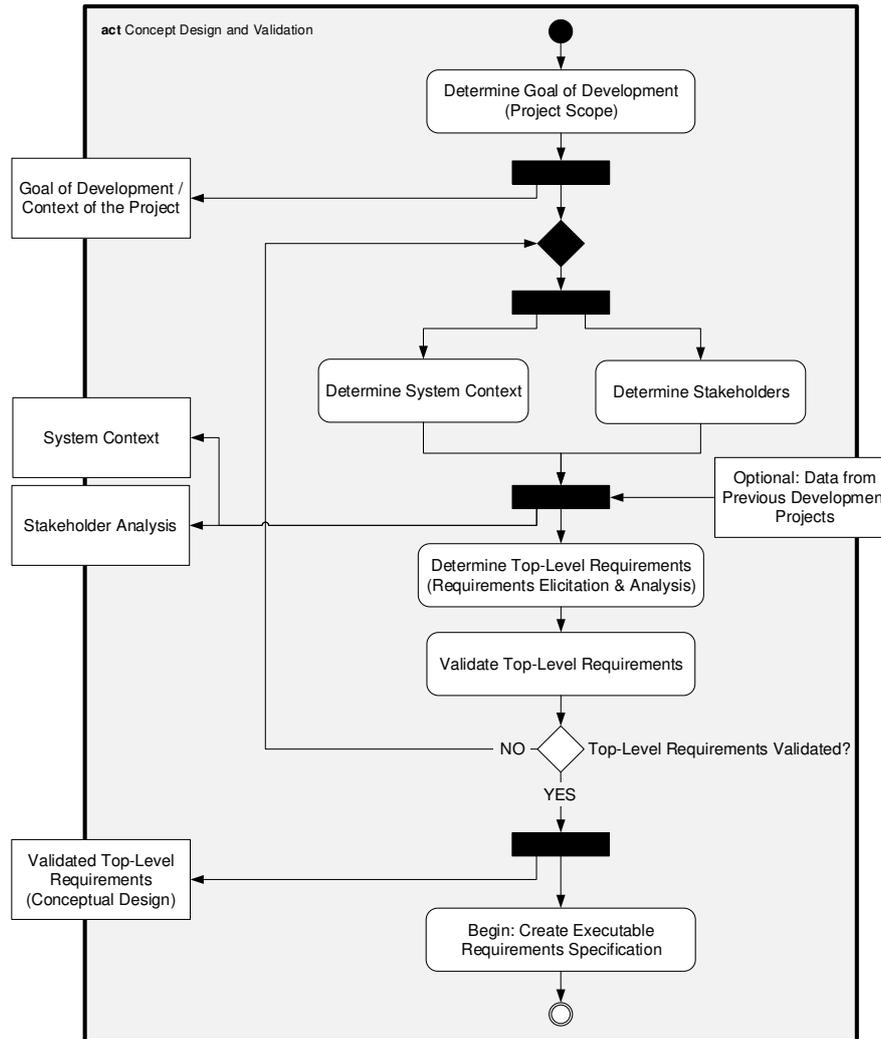


Figure 4.12 – Begin of system development: Concept design and validation process

4.2.1 Project Scope, System Context and Stakeholders

At the beginning of system development, the scope of the project is determined for the system under design (SuD). This includes the definition of a specific goal of development which reflects the intended purpose and application (global mission) of the overall product. At this point, it is also important to determine what goals shall not be pursued in the course of development. For the determination of system context and stakeholders, use case diagrams of the UML and SysML are used. During the process of project scope definition, it is also possible to perform market and preliminary feasibility analyses. This is useful in order to analyze the scope of the SuD more closely or to determine additional services and quality properties for the SuD based on similar system designs or technological advancements. Market and feasibility analyses also help to determine new system context information and additional stakeholders.

System context is an accumulation of different elements of reality that have an influence on the SuD. The analysis and limitation of system context is important in order to determine a bounded set of requirements for the SuD [268]. System context

can be derived from the inspection of possible interfaces or interaction points that exist between the SuD and other entities. This includes other systems, sensors, actuators or environmental influences.

The term stakeholders comprises all entities that are, directly or indirectly, related to the SuD. This includes humans as well as objects or institutions. The process of stakeholder identification and analysis is, alongside with system context analysis, crucial in order to identify all sources of information for the development of design concepts and requirements specifications [365]. On one hand, direct involvement of stakeholders in the design process can be complicated since different groups involved in the development of a system interact with different stakeholders on different levels. On the other hand, stakeholders may provide direct feedback on early design decisions or help to identify other sources of knowledge, e.g. system interfaces and yet unknown stakeholders. At this high level of system development, stakeholders are usually identified and analyzed with respect to the goal of development and the resulting intended behavior and quality of the SuD. In terms of system security however, it is useful to specify misuse stakeholders with potentially harmful intentions for system misuse, including system manipulation or sabotage.

The determination of system context and stakeholders needs to be validated against the overall goal of development in order to make sure that no important source of requirements is missing during requirements elicitation. Validation against the overall goal of development is also used to identify system context and stakeholders that are unnecessary for the SuD.

4.2.2 Top-Level Requirements Elicitation and Analysis

After completing the processes of system context and stakeholder analysis, top-level requirements need to be determined and analyzed. At the beginning of this process, global services, objectives and constraints for the SuD are determined. In order to do so, specific needs for each stakeholder need to be identified with regard to the intended system application and system context. Operational services, non-functional objectives, i.e. quality properties, and constraints contribute to the overall application or mission of a SuD. This means, that the SuD is considered to provide a set of certain services with specific quality with defined system context for a set of different actors, i.e. stakeholders. Different methods for determining requirements exist. These include observation, questioning and creativity techniques [268]. In this work, a use case and scenario-driven conceptual design process is applied with associated requirements documentation (cf. chapter 2.3.4.1). This is done in compliance with the mission level design approach (cf. chapter 2.3.4.2).

At the begin of requirements elicitation and analysis, stakeholder needs and requirements are often expressed abstract or vague and need to be explored further. Thus, this stage of design is characterized by a high amount of creativity and constant change. Moreover, this process is carried out in collaboration between different stakeholders with different backgrounds, knowledge and intentions e.g. customers, marketing staff and system designers [373]. Therefore, the identification and specification of use cases, qualities and constraints is carried out in the form of so-called essential use cases. In contrast to concrete use cases described in chapter 2.3.4.1, Constantine and Lockwood [88] describe essential use cases to be an abstract and generalized way to describe stakeholder intentions independent from technology or

implementation considerations. Weilkiens [365] considers the usage of essential use cases with SysML useful to provide an adequate level of abstraction during early design stages in order to achieve common understanding and consensus between all parties involved. Questions to be answered include: “*What services and functions shall the SuD provide and wherefore?*” and “*With what qualities shall services be provided?*”. Using essential use cases at the begin of development strongly supports the elicitation of customer related requirements, e.g. when using Quality-Function-Deployment (QFD). QFD was developed by Akao [7] in order to improve design quality of each development stage by focusing on customer needs, starting with product planning. Customer requirements are established by differentiating between e.g. customer requirements and technical characteristics or solutions. This is done in order to ensure that the true needs of customers can be captured [7].

Each essential use case has a relation to at least one stakeholder. Use cases with no affiliation to any stakeholder should be avoided. If no stakeholder has an interest in the service determined by a use case, this service is likely to be non-essential for the final product. In the context of this work, essential use cases begin with the preliminary phrase “*provide*”. They may contain multiplicities and directed connections. In addition to services that are described by essential use cases, non-functional properties and requirements are added in the form of annotations. In parallel to the determination of essential use cases, the identification and creation of essential misuse cases is started. This is important in order to identify top-level system security use cases early during design. In this case, the principle behind essential use cases can also be used. In order to counter possible system misuse as defined by misuse cases, essential security use cases are developed. Security use cases constitute a specialization of system use cases since they originate in system misuse analyses. In the context of this work, security use cases have the same semantic characteristics as essential or concrete use cases.

Misuse cases are derived from the analysis of misuse stakeholders. This can only be done by involving security specialists within the design process since potential misuse stakeholders cannot be directly accessed. In many cases, misuse stakeholders can only be defined indistinctly as surrogate for a group of potential security threads. System security specialists rely on experience and well documented threads for their specific field of knowledge (domain). Thus, they are able to analyze already defined essential and concrete use cases in order to derive essential or concrete misuse cases and specific security use cases to counter potential misuse threads. Since misuse cases and related security use cases are determined by analysis of specified use cases, misuse cases are typically found on lower hierarchical levels, e.g. as part of superordinate top-level use cases. At this point it is important to note, that system security issues also evolve later during development, e.g. from specific system architectures, interface definitions, unit specifications and implementation decisions. Therefore, these issues need to be identified with security experts, e.g. via security audits, during later development stages. In that case, this information may be used as feedback for iteration on top-level design requirements.

Another contribution to this stage of development comes from knowledge and experience from previous development projects similar to the SuD. This knowledge should be re-used, if available, in order to speed up the design process and gain higher maturity. In the aerospace industries for example, it is good practice to

re-use reliable and approved system functions and components from earlier aircraft developments that have already been validated and verified. In addition, technology reviews of other system providers supply a good source of information for needs, services or other requirements that may not yet be known to stakeholders. This is sometimes related to as excitement needs, attributes or factors, a term used in the Kano model to describe unexpected or latent needs of stakeholders that lead to a high level of satisfaction [176]. Excitement attributes are, for example, introduced to system customers by system engineers. Data from similar developments may also be useful in order to identify additional stakeholders or system context. At this level of abstraction, use cases often become more concrete and do not include a “*provide*” prefix. This can be an indication for design re-use, i.e. the integration of already known and validated components, e.g. use cases used in previous developments.

In order to document, structure, manage and refine top-level requirements to create a conceptual design, mind maps have proven to be adequate [130], [129] and [373]. With regard to the process of top-level requirements elicitation depicted Figure 4.12, the conceptual design in the form of a mind map becomes the central result of the overall process. Thus, the creation of a conceptual design mind map commences with the start of development in order to document, unite and manage results from each process shown in Figure 4.12. Mind maps are diagrams used to gather, organize and access information visually. In general, mind maps connect ideas as well as concepts to a central point of interest, e.g. an idea or concept. Also, mind maps are used to document or plan a subject with clear structure. The term and method behind were described in detail by Buzan [70]. In many areas, mind maps are considered a cognitive tool for information processing that enable users to exploit their full potential of information processing [209]. In principle, mind maps are similar to semantic networks or cognitive maps [363]. In terms of requirements engineering, mind maps are an established tool for requirements elicitation and analysis, especially to express dependencies and to document refinements. Mind maps are also used as supporting tool for creativity techniques such as brainstorming [268]. In the context of model-based engineering and agile software development, mind maps have also been successfully used [363] and [209].

Dedicated mind mapping tools often provide the possibility to export different file types for created mind maps. A mind map can also be colored, enhanced with icons and figures or complemented with connection lines between nodes in order to document dependencies, priorities or to cluster requirements. Moreover, nodes of a mind map can be linked to other nodes or external files that can directly be opened from the mind map.

As the process of requirements elicitation and analysis progresses, top-level requirements are decomposed top-down with increasing level of detail. Decreasing the range of possible operational scenarios for the SuD is important in order to delimit the design space with regard to the finite set of true stakeholders needs. If decisions at this level are postponed to later design stages, it is possible to obtain functional and non-functional properties for the SuD that are not intended by stakeholders. This is because at some point during development or implementation, a group or individual will have to take a concrete decision in order to implement a feasible solution for the SuD. In doing so, a high risk exists that the decision taken is not based on sufficient knowledge of system context and the overall mission, thus leading to

possibly unknown design flaws. As a result, essential use cases need to be refined or complemented with more specific use cases or misuse cases as early as possible during design.

For the purpose of design formalization, nodes describing essential use cases are equipped with direct links to the respective model created in UML or SysML. These models contain information on the use case, its actors and system context. It is also possible to describe specific architecture or technological requirements for the SuD as part of an essential use case model. This can be useful in order to define system architecture or technology criteria that are essential for the overall design. Alongside with nodes that describe services of the SuD in the form of use cases, other nodes can be created that are used to describe top-level procedures, interactions, or dependencies. These nodes may be described with other diagram types of the UML or SysML, e.g. activity or sequence diagrams. However, it is noteworthy that other diagram types should use the same high level of abstraction than essential use cases. Otherwise, the level of technical detail may be counterproductive since not all stakeholders will be able to understand the domain specific knowledge provided.

Use case and misuse case nodes can also be equipped with non-functional requirements, often referred to as quality or performance requirements. This can be done by annotation of use and misuse diagrams or by introducing specific child nodes for the description of qualities or constraints. In addition, standalone non-functional requirement nodes may be defined that specify top-level quality requirements or constraints for the SuD. Moreover, descriptions, annotations or comments can be added at lower hierarchical levels of a conceptual design mind map. These nodes can contain descriptions in the form of text, graphics or external links to other objects including tables, audio files or preliminary architecture descriptions in the form of computer-aided design (CAD) files. This allows system designers to be as creative as possible during this stage of system development.

Figure 4.13 depicts an example of a conceptual design mind map with different objects linked to different nodes. Nodes describing essential use or misuse cases are linked to UML or SysML models that determine the respective use or misuse case in detail (bottom node, “Provide X.3”). In order to enhance the description of a use case, information or conceptual data models can be created and linked to any use case node. UML tools like *Visual Paradigm* directly support combined use case and data modeling by means of sub-diagrams linked to use case nodes [205]. It is important to gather and determine all necessary information needed by the SuD in order to provide a certain use case as early as possible during design. Conceptual data models can be created, for instance, by using Chen notation as part of entity–relationship (ER) models, Object Role Modeling (ORM) or UML class diagrams.

This formalization process strongly contributes to the development of a consistent data model during later design stages, e.g. for the development of a formal executable system specification. In Figure 4.13, the use case “Provide X.3” is linked with a conceptual data model by means of an ER sub-diagram. Links to other objects, e.g. tables or graphics, allow to describe non-functional requirements, specific conditions or constraints in more detail. This can be done with any tool and at any level of abstraction suitable for the requirement under consideration. In Figure 4.13, one use case node is enhanced with an additional table that specifies priority aspects of the related use case in more detail (upper node, “Provide X.1”). Node “Provide X.2.1”

is complemented with an external document while node “Provide X.2” has a child node that contains a graphic which specifies a user interface design requirement.

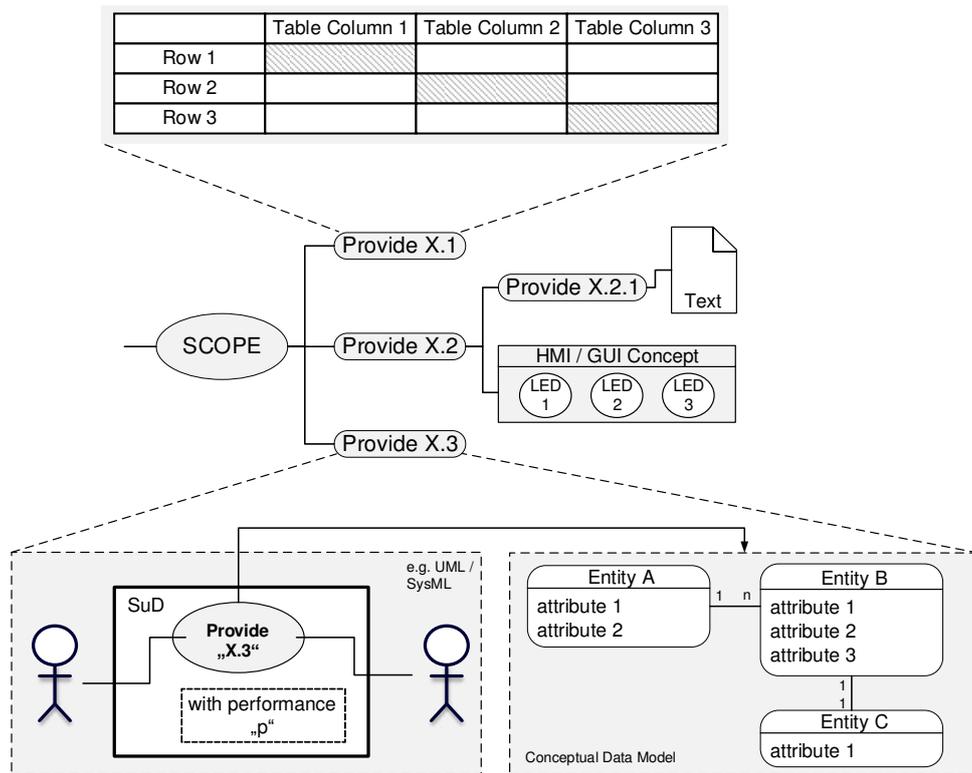


Figure 4.13 – Conceptual design mind map with nodes linked to different external objects (grey), including UML/SysML models, conceptual data models, tables, graphics and texts

By using mind maps for the conceptual stage of system design, it is possible to create an abstract conceptual requirements description that unites use cases, constraints as well as quality properties for the SuD. Top-level requirements descriptions in terms of use cases, misuse cases and quality requirements linked within a conceptual design mind map can, however, not be used to analyze and validate the behavior of the coupled and integrated overall system [130]. Therefore, the results of this development stage rather constitute the basis for the development of an executable requirements specification that is used for specifying a specific system design and for validation by means of computer-based system simulation. However, mind maps that have been created at this stage of development will continue to be used for documentation and requirements management. Thus, they are extended step-by-step as the development process proceeds.

4.2.3 Concept Validation

The phase of conceptual design is characterized by constant change and iteration. Although functional and non-functional requirements are identified and documented in mutual collaboration between different stakeholders, it is important to independently validate results of this phase. Therefore, a differentiation is made between system designers who develop a conceptual system design and validators who validate and test design requirements with regard to stakeholder needs. Furthermore, validators need to document the validation progress and provide feedback to system designers. The importance and benefits of independent validation as well as

verification has been published in many areas of systems engineering, e.g. for software systems [22]. Concept validation is a major prerequisite for the creation of an executable requirements specification. During this stage of design, two major quality criteria analyzed are completeness and testability. Other criteria including consistency and feasibility are analyzed as well but will become focal points during validation by means of executable specifications. More information on each of the specific specification quality criteria can be found e.g. in reference [45].

It is important to establish a concept design foundation with adequate fitness in order to provide services and qualities satisfactory to a set of product stakeholders. This is done by critical assessment of the conceptual design in collaboration between representatives for each stakeholder and engineers managing the validation other than the design engineers. During the validation process, requirements can be prioritized or classified according to requirement levels, e.g. in accordance with *Request for Comments 2119* (RFC 2119) [53]. Thus, system designers are able to determine whether a requirement is indispensable or optional. This information is used in order to find a feasible design in the course of development.

Each requirement is analyzed in terms of relevance for the overall design, based on the initial purpose for the SuD. Moreover, only use cases and quality properties that contribute to the satisfaction of stakeholder needs are regarded valid. Requirements need also be assessed in terms of proportionality with regard to non-functional criteria including overall system cost or technical feasibility. The process of validation is performed in the form of critical design assessments, e.g. during single stakeholder reviews, focus group meetings and workshops. In addition, other manual validation techniques may be used. See reference [45], for instance, for more information on manual validation approaches.

Security related requirements described in the form of misuse cases need to be validated with security experts. When involving customers in the validation process of misuse cases, a change of initial use cases may become useful in order to counter or avoid specific misuse cases by design, thus remedying possible security risks. A preliminary safety assessment can be performed in parallel to concept validation in order to determine system safety relevant information. As part of this process, new functional and non-functional requirements may emerge. Necessary changes determined during validation are fed back to earlier stages of design in order to find design compromises between the different stakeholders. Because a min map is used as central element of documentation and requirements management, relations between different nodes can be re-structured during validation if required. Changes require a re-validation of the overall concept design. In practice, several iteration loops are performed already in parallel with the beginning development of the executable requirements specification. In summary, concept validation is used to ensure that the specified SuD does what it is intended to do (function) and how it will do it (performance and quality). In addition, validation is used to minimize the number of properties of the SuD that stakeholders regarded as unwanted or unnecessary.

4.3 Executable Requirements Specification Development

After successful top-level requirements elicitation, conceptual design and validation, the phase of requirements analysis and executable requirements specification (ERS)

development is started. This phase is characterized by the development of a formal, complete, consistent and feasible specification model of requirements that reflects the needs of all stakeholders. On the one hand, an ERS provides the basis for the development of an executable overall system specification. On the other hand, the development of an executable requirements specification is pivotal for the development of automated and interactive validation processes. An executable requirements specification can also be regarded as a more detailed, concrete and executable version of the conceptual design. Executable requirements specifications consist of two general aspects. Firstly, an integrated and executable overall mission model that unites driver and evaluation models for the system under design (SuD), and secondly, an executable human machine interface (HMI) concept model. Both types of specification models determine requirements for the SuD and are used to validate system specifications during the following stages of development. In general, an executable requirements specification is defined as follows:

Definition 8 *An **Executable Requirements Specification (ERS)** consists of two types of models that are used to determine design requirements (functional and non-functional) and constraints for a system under design (SuD). The first part of an ERS is called overall mission model. It determines the operational concept of the SuD together with integrated service, quality, configuration and customization requirement models. The second part of an ERS is a human machine interface concept model that is used to determine usability requirements. Both models are executable in a simulator. The overall mission model can be executed in combination with executable system specifications for automated validation purposes while the HMI concept model can be used in combination with executable system specifications to form virtual prototypes.*

An executable requirements specification can be described by a 2-tuple (OM, UM) where:

- OM = is an executable workflow that determines the overall mission of a system under design; It consists of a finite non-empty set of mission models $\{mission_1, mission_2, \dots, mission_n\}$ with specific order that are also created in the form of executable workflows
- $UM = \{um_1, um_2, \dots, um_m\}$ is a finite non-empty set of executable HMI concept models

In contrast to top-level requirement models established during conceptual design, e.g. by means of essential use cases, misuse cases and mind maps, an executable requirements specification is characterized by formal computer models that can be executed in a simulator. Most computer models require formal and explicit procedural instructions in order to simulate more complex processes and coupled system behavior. That implies, that essential use cases and other top-level concepts need to be made more specific and concrete as part of the work of system architects and system designers. During this process, an analysis of each top-level use case is performed together with other stakeholders to narrow down the design space for the SuD. In the following sections and sub-chapters, the terms use case, misuse case

and security use case are jointly referred to as use case or atomic missions since they underlie the same design principles. This is also intended to facilitate reading. Detailed definitions for each term are provided within the respective sub-chapter.

Figure 4.14 depicts an iterative and phase-oriented process for executable requirements specification development and validation. Each activity of the process (ovals) receives information from previous activities or inputs from activities prior to this stage of design (rectangles on the left and right). Activities generate outputs (rectangles on the left) in the form of computer models. Activities containing a fork symbol consist of a set of sub-activities that will be elaborated in the following chapters.

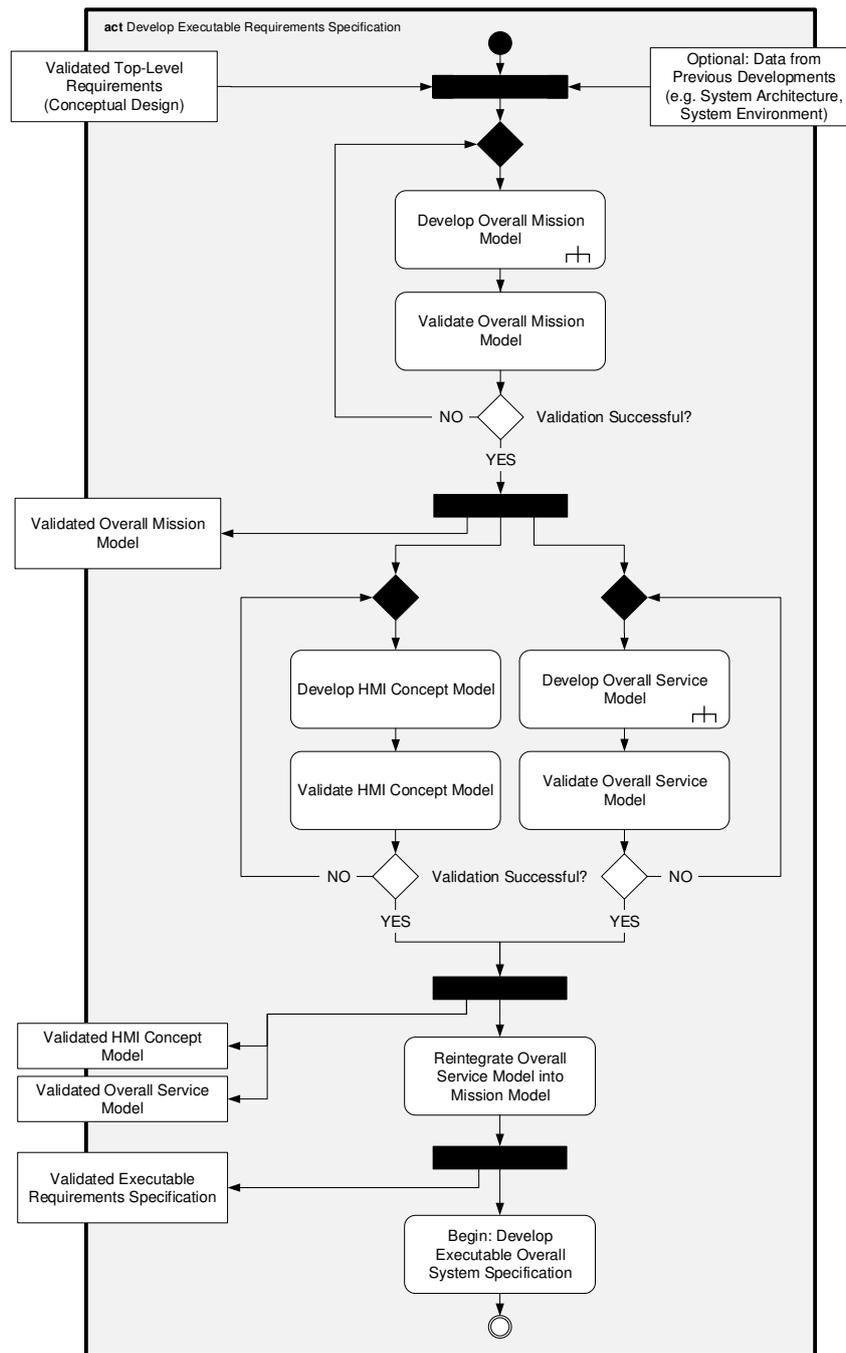


Figure 4.14 – Process for executable requirements specification development and validation

The overall process commences with the development and subsequent validation of an overall mission model that reflects the intended application, performance and typical use of the SuD in the form of one or more operational scenarios. Inputs for both activities are top-level requirements determined during conceptual design as well as information and components from previous developments. In the field of aircraft development for instance, general categorization and design of subsystems, functions and system architecture as well as system environment remain stable over different generations of aircraft. This provides the basis for the development of a feasible system architecture for current SuD. Data from previous or similar system developments is useful for determining system interaction points and data models needed for each atomic mission. Moreover, configuration and customization possibilities can be derived that are based on experience. Principal output of this design phase is an overall mission model that unites missions (operational system scenarios), the design of atomic mission models (i.e. use cases), atomic mission-specific data models, system interaction models as well as mission-specific configuration and customization models for the SuD.

Following the determination of an overall mission model, including system interaction, data and configuration models, a concept for HMIs of the SuD is developed and validated in order to include usability aspects early during design. Specific emphasis is put on complex graphical user interfaces (GUIs), e.g. multipurpose HMIs, including touch screens or other interactive human operated system controls or information monitors. In parallel, integrated service models are developed and validated for each atomic mission model (i.e. scenarios for each use case). This means, that essential, alternative and optional procedures are developed for each atomic mission. Thus, services provided by the SuD are specified in the form of concrete process sequences with certain characteristics for each atomic mission. As part of this process, mission-specific interaction and data models are refined or extended iteratively. Atomic missions as well as service scenarios describe functional and non-functional requirements for the SuD. See chapter 2.3.4.1 for more information on mission and scenario-driven system development.

Eventually, overall mission and service model are re-integrated, thus completing the executable requirements specification. Although validation against concept design has been performed sequentially after each activity, the integrated executable requirements specification can again be validated against top-level requirements from concept design. In the case of successful validation, the process of system design continuous with the development of an executable overall system specification. The development of overall mission and service model follows the meet-in-the-middle paradigm that is used as part of the mission level design approach (cf. chapter 2.3.4.2). This means, that the design process is performed top-down while integrating bottom-up developed model components from previous development projects. By this, designers can either create new mission or service models or use existing and already validated library components. In the following chapters, each step of the development of an executable requirements specification is elaborated more closely. The process of validation, as part of the overall process depicted in Figure 4.14, will be elaborated more closely at the end of this chapter within sub-chapter 4.3.8. As for the process of concept design, the process depicted in Figure 4.14 is considered part of an iterative and phase-oriented development process, e.g. the V-Model described.

4.3.1 Overall Mission Model Development

The development process for an overall mission model is divided into five major steps as depicted in Figure 4.15. Top-level requirements from concept design as well as additional data from previous development projects are used as inputs for each of the steps shown. Starting top-down, the overall process commences with the development of a mission model during step one. This model represents a typical operational scenario for the system under design (SuD). Subsequently, three parallel activities are started that interact with each other and result in the development of an integrated overall mission model.

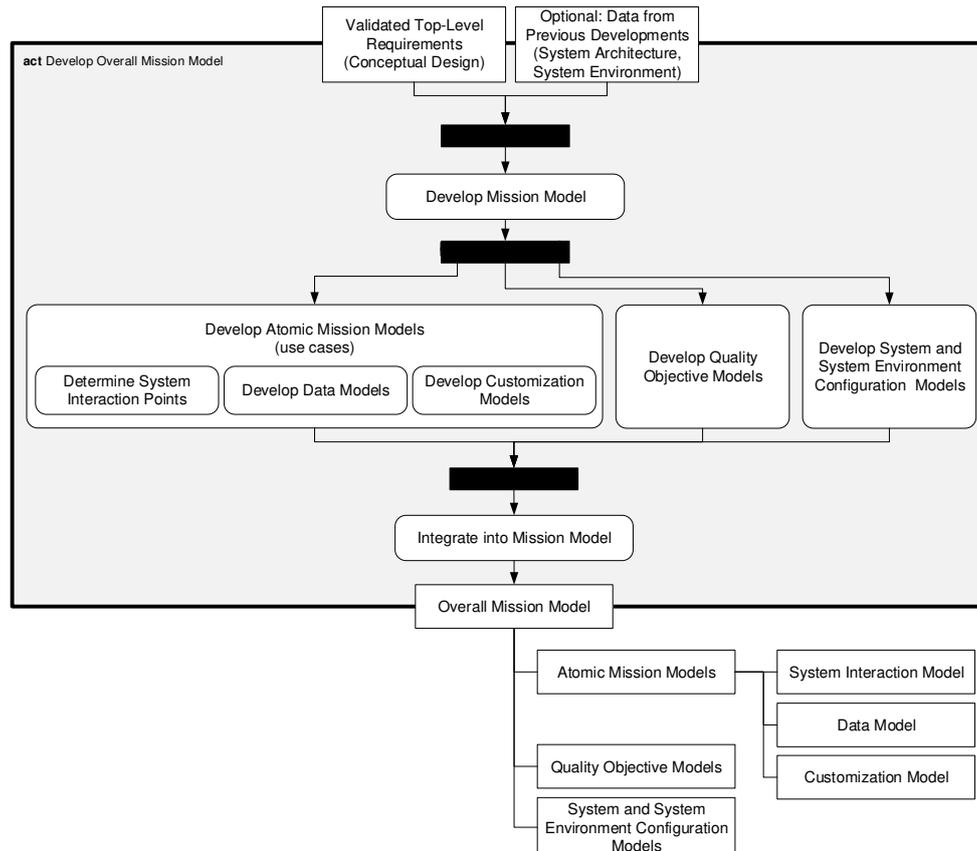


Figure 4.15 – Process for overall mission model development

Atomic mission models, i.e. use cases combined with non-function requirements, are derived from top-level design (essential use cases). For each atomic mission, necessary system interaction points are determined together with specific data models and customization possibilities. Quality objective models determine global non-functional requirements for the system under development at mission level. At the same time, parameters for system and system environment configuration are determined in relation to each of the atomic mission models. These parameters strongly influence the functional and non-functional characteristics of the respective atomic mission model. Thus, system interaction models, customization as well as mission-specific data models may need to be revised in order to account for impacts from system and system environment configuration. During the last step of overall mission design, all submodels are integrated back into the top-level mission model. All models that are developed as part of overall mission model development can be

linked to the original mind map containing the conceptual design. By this, models can be directly accessed from the mind map and requirements remain traceable throughout the design process. All activities of overall mission model development depicted in Figure 4.14 are described in detail within the following sections.

Liebezeit defined a mission to be the description of a typical operational scenario for a system [198]. In contrast, Schorcht states that each system has a set of operational requirements that are defined in the form of different use cases called missions. In the context of this work, the definitions of Schorcht and Liebezeit are adapted in order to define a more distinguished classification for mission models in the context of executable specifications. The following definition for mission models is derived from Liebezeit's definition while the definition for atomic mission models in chapter 4.3.2 is derived from Schorcht's perspective on missions.

Definition 9 A *mission model* is an executable overall operational scenario for a system under design that includes system services in the form of sub-missions or atomic mission models, system quality objectives as well as system and system environment configuration models.

Thus, a mission model represents one typical case of the intended high level application of a system in service together with quality requirements and aspects of system environment, i.e. an operational concept. In other words, missions express desired behavior and quality in terms of how the systems is intended to be employed and used together with expected changes of the environment in which the system is intended to be integrated. Missions are also used to determine an overall system specification model during later design stages and to validate the overall system specification by means of simulation. Moreover, mission models can be re-used to generate system test case scenarios for test bench integration. A typical case of an operational scenario is determined by sub-system mission models, atomic mission models, quality objectives as well as system and system environment configuration steps with specific order. Figure 4.16 depicts the process of overall mission model development in relation to the development and validation of a system under design.

Mission models are created in the form of directed executable graphs and are composed of mission models at lower hierarchical level, atomic mission models, quality objective models, system and environment configuration models, control flow elements as well as transitions (edges). Mission models (MM) constitute a task-specific modification of executable workflows. They can be described by a 6-tuple $(M, C, D, P_{MM}, I, c_n)$ where:

- $M = \{m_1, m_2, \dots, m_i\}$ is a finite non-empty set of mission nodes
- $C = \{c_1, c_2, \dots, c_n\}$ is a finite non-empty set of control nodes
- $D =$ is a finite set of shared memories / data storage modules ($\emptyset \in D$)
- $P_{MM} =$ is a finite non-empty set of configurable parameters
- $I \subset C =$ a finite non-empty set of initial control nodes
- $c_n \in C$ is the final node

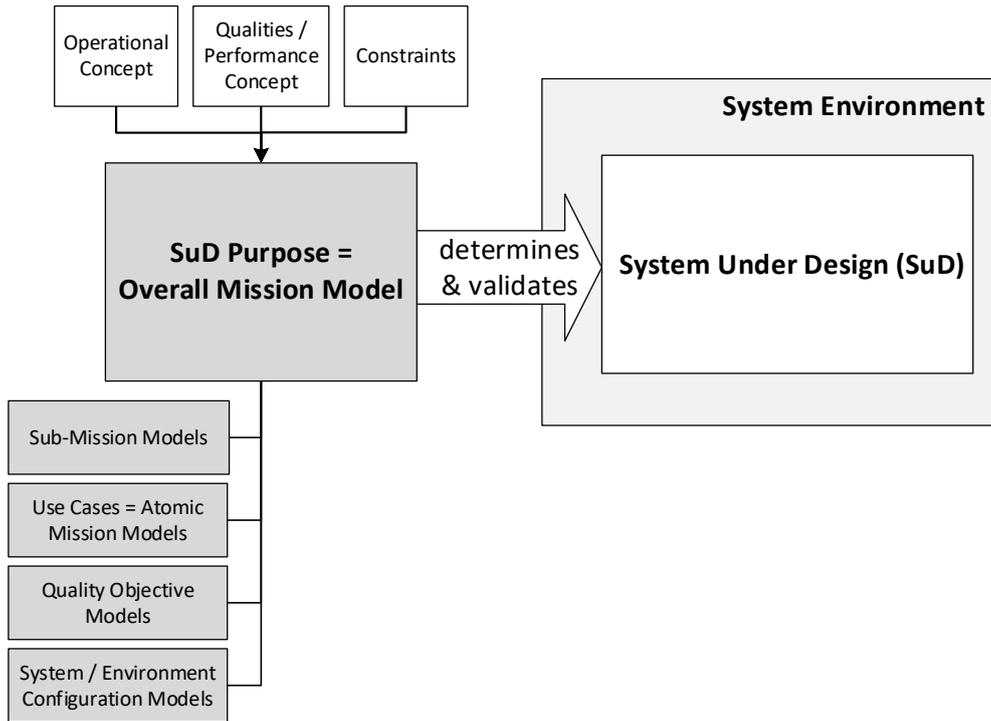


Figure 4.16 – Overall mission model development in relation to the development and validation of a system under design (SuD)

Mission nodes may only be chosen from a finite set of node types: $\forall m \in M: m \in \{MIS, AMIS, QO, ENV\}$ where:

- *MIS* is a finite set of sub-mission models on a lower hierarchical level. Any mission model on a higher hierarchical level $mm_x \in MM_L$ must not be part of any mission model on a lower hierarchical level $mm_y \in MM_{L-i}$. $MM_L \cap MM_{L-i} = \emptyset$
- *AMIS* is a finite set of atomic mission models (use case models)
- *QO* is a finite set of specialized atomic mission models, called quality objective models, exclusively used to describe non-functional system requirements
- *ENV* is a finite set of system and system environment configuration models (used in relation with atomic mission models)

It is possible to determine different mission models for one system in order to provide different operational scenarios, e.g. for different system customers. In the case of aircraft, different airlines have different operational scenarios for different groups of aircraft. Airline-specific operations are determined, for instance, by airline-specific business models or the intended route of an aircraft, e.g. short range, mid-range or long range operation.

Mission models, as part of executable requirements specifications, are used as starting point for development of sub-mission models, atomic mission models and quality objective models. After the development of all submodels of a mission model has

been finished, these submodels are integrated back into the original mission model to form an overall system model. Eventually, service models of the subsequent development step are integrated into atomic mission models of the overall mission model in order to create an executable requirements specification. This specification is used to develop and validate a feasible system design, i.e. an executable overall system specification. In the context of this work, mission models are also referred to as operational process models or validation flow models. The term validation flow relates to the pivotal function of mission models as part of an automated system validation process. During validation, executable requirements specification and executable system specification are executed jointly within an executable validation model. In other words, overall system execution is orchestrated by the overall mission model. In the course of this process, the mission model determines control as well as data flow between the two types of executable specifications. More information on the process of automated validation is provided in chapter 4.4.2.

Control nodes, including initial nodes and the final node, shared memories as well as configurable parameters are used according to the description of executable workflows in chapter 4.1. In addition, four global parameters were added for documentation purposes. Parameters “*Associated_System_Name*” and “*Associated_System_ID*” are used to identify the system that is associated with a specific mission model while parameters “*Mission_Label*” and “*Mission_ID*” are used to identify the respective mission model. Control and data flow properties are also use in accordance with the principles of executable workflows (cf. chapters 4.1.1 and 4.1.2). Figure 4.17 depicts an example mission model with different types of mission nodes. As with executable workflows, mission models are created by determining a flow of activities. This is done in order to structure the execution of different submodels, e.g. atomic mission models.

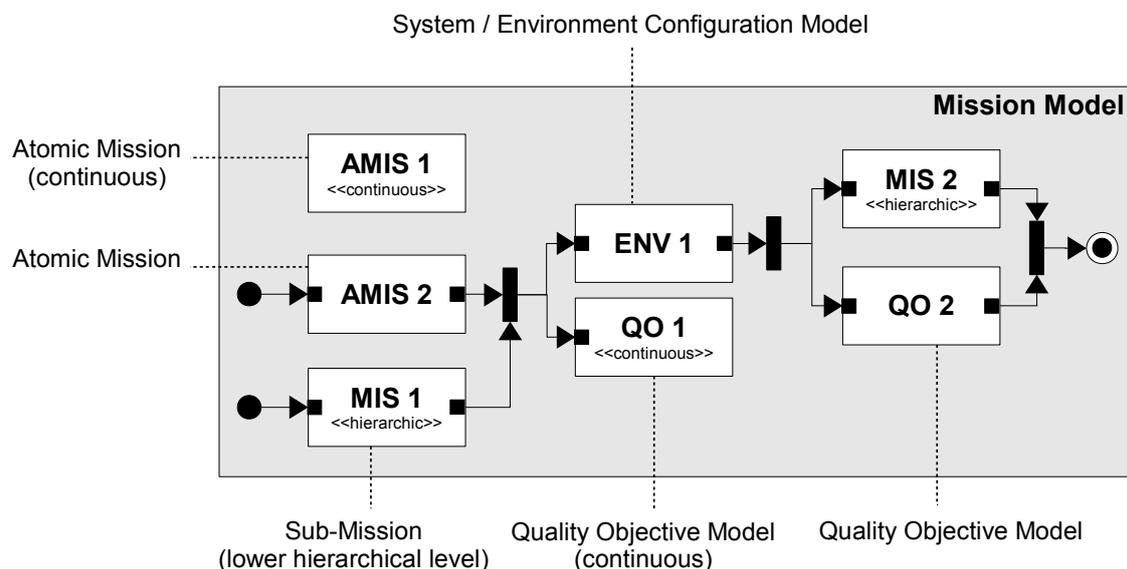


Figure 4.17 – General mission model structure example

Sub-mission models describe mission models at lower hierarchical levels. They are used to group task-specific sub-missions for any given top-level mission model rather than to describe an overall operational scenario at overall system level, e.g. in order to coordinate different complex sub-system missions for a system of systems. Since sub-

mission models are embedded in the control flow model of a superordinate mission model, they have an additional input and output port. Receiving a token at the input port will trigger a sub-mission model active. Thus, initial control nodes of sub-mission models will only fire after receiving a token at the input port. When completed, i.e. when a token is received by a sub-mission final control node, an output token is generated at the output port to hand back control flow to the superordinate mission model.

In contrast to mission models, each atomic mission model determines one specific use case for the SuD. Moreover, each atomic mission model contains a use-case specific system interaction, data and customization model. Later during design, a service model is developed for each atomic mission model that includes functional and non-functional design requirements for the system under design (cf. chapter 4.3.5). While use case-specific quality requirements like timing requirements are directly included within the service model of each atomic mission model, global non-functional system requirements or constraints including cost, energy consumption, or weight are directly modeled at mission level. This is done via quality objective models. A quality objective model can be regarded a specialized atomic mission model that only describes non-functional system requirements, i.e. quality objectives. Each quality model determines a range for any given non-functional requirement instead of a single value.

According to Liebezeit [198], mission models include a mission target as well as specific configurations for system and system environment. In the context of this work, a mission target can be regarded as a composition of sub-mission models, atomic missions as well as quality objective models. These models determine functional and non-functional requirements as part of a model with directed process flow in order to achieve a specific operational scenario. However, in the course of execution of atomic mission models or quality objective models, the SuD or its environment may be required to be in a specific state. In this case, system or system environment states may be expressed by specific configurations of parameters [198].

State changes affect logical as well as physical parameters. Within the field of aircraft for example, system configurations include the configuration of subsystem states, e.g. the retraction of landing gears or a specific fill level of potable water tanks. Typical environment configurations above aircraft level include flight phases or ground service parameters that, in result, will affect several subsystem state changes at once. Providing a specific system or environment configuration can be considered a precondition for other elements of a mission model. Within mission models, system and system environment configurations are determined with dedicated system and environment configuration models. In contrast to system and environment configuration models that are required as part of an operational scenario, customization models are integrated within atomic mission models. Customization models are used to provide different configuration possibilities for each use case. This includes functional as well as non-functional parameters.

Within mission models, mission flow starts at one or more initial control nodes and ends with at one final control node as depicted in Figure 4.17. Initial, final as well as other control nodes are used to determine a specific control flow for each mission model. With this, other mission model nodes can be arranged in order to be executed in parallel or in sequence. In the course of mission model design,

sub-mission models, atomic mission models, quality objective models together with system and system environment configuration models are structured according to their intended usage determined during concept design. In some cases, a sequential arrangement of atomic mission models is used to determine that different use cases are not intended to be performed in parallel or that they cannot be performed at a time, e.g. because of physical circumstances. Also, different atomic mission models may be dependent on each other so that a certain atomic mission model sequence is required in order to provide a specific overall mission behavior.

In other cases, it might be crucial to provide the SuD with capabilities to execute different use cases in parallel. In this case, atomic mission models as well as other mission nodes are considered independent from each other in a way that none of the parallel mission nodes requires an action or information from any of the other nodes in order to execute successfully. However, parallel mission nodes may still interfere with each other, since they all interact with the SuD. Although this is intended, emergent behavior might be caused. Emergent behavior is analyzed and remedied as part of validation, verification and test processes.

As with executable workflows, mission models can also contain detached or parallel mission nodes. A detached mission node can be used to model mission nodes that are executed over the entire mission (cf. chapter 4.1). Detached nodes without input port begin execution after initial control nodes have started firing and finish execution when the control flow reaches the final control node. In Figure 4.17, atomic mission node *AMIS 1* represents a detached node. Detached mission nodes can be used to model continuous atomic mission models (continuous use cases) and quality objective models or to continuously alter different system environment parameter configurations. Continuously executing mission nodes can also be integrated at specific points of a control flow graph of a mission model. In this case, the respective node starts executing after reaching a certain point of mission model execution and finishes execution together with the overall mission model.

Following the design of a global mission model, all mission nodes need to be specified in more detail. Within the next sub-chapters, the development of atomic mission models, quality objective models as well system and system environment configuration models will be described in detail. After detailed design of all required mission nodes, all submodels are integrated back into the respective mission model, thus forming an overall mission model for the system under design.

4.3.2 Atomic Mission Model Development

Atomic mission models (AMIS) represent an essential part of mission models. They form the foundation of an executable requirements specification and are derived from essential and concrete use cases of the conceptual system design. It is the aim of this design step, to bridge the gap between concept and system requirements and to define services of the system under design (SuD) more closely. As part of this work, the following definition shall be used:

Definition 10 *An **atomic mission model** represents one specific use case with associated functional and non-functional qualities that is to be provided by a system under design. Atomic mission models unite use case-specific actor, system, and system context interaction, data as well as customization models.*

In comparison, atomic mission models determine specific services provided by a system while mission models determine scenarios for the application of different services during the operation of a system. The fundamental structure of atomic mission models is based on use case diagrams of the Unified Modeling Language (UML) and Systems Modeling Language (SysML). As a result, essential components of an atomic mission model include a central use case node that contains a service model as well as a set of actors associated with the respective use case node. Thus, atomic mission models describe operations and processes between actors, system and system environment in order to achieve a certain goal (cf. chapter 2.3.4.1). An atomic mission model (AMIS) can be described by a 10-tuple (ACT , UC , $IAMIS$, P_{AMIS} , P_C , P_D , EV , EM , IN , OUT) where:

- ACT = is a finite non-empty set of internal or external actors
- UC = is the central use case node, containing a service model
- $IAMIS$ = is a finite set of included atomic mission models, representing included use cases (optional)
- P_{AMIS} = is a finite non-empty set of configurable parameters
- $P_C \in P_{AMIS}$ = is a global parameter containing a use case-specific customization model
- $P_D \in P_{AMIS}$ = is a global parameter containing a use case-specific data model
- EV = a set of two event elements
- EM = a set of two event control modules
- IN is the control flow token input port of the overall model
- OUT is the control flow token output port of the overall model

Figure 4.18 depicts the basic structure of an atomic mission model. Since atomic mission models are part of a superordinate control flow that is described as part of a mission model, atomic mission models have an input port *in* (triangle on the left-hand side) as well as an output port *out* (triangle on the right-hand side) for receiving and creating control flow tokens. Within Figure 4.18, control flows are symbolized by lines with spherical ends while data flows are symbolized by lines with triangular arrows. Logical links between elements of atomic mission models are symbolized by dashed lines. Upon receiving a control flow token at the input port IN , an event control module (EM) is triggered (*Start Trigger*). This module is linked to an event element (EV) called *Start Event*.

Events are shared model elements that are used e.g. within timed simulation domains to create asynchronous events during simulation. They can be linked to other model elements, e.g. composites and finite state machines. Within atomic mission models as well as in all executable workflow related models, event elements are used to model control flows. By linking an event element to another model element, that element gains control and starts executing as soon as the respective event element

fires. When becoming active, the module *Start Trigger* triggers an asynchronous event for *Start Event* that is initiated immediately, i.e. quasiparallel. The central use case node of an atomic mission model contains two internal event elements as part of a hierarchical submodel. One of these elements is linked to the event element *Start Event* while the other one is linked to the second event element of atomic mission models *Stop Event*. Upon firing of the element *Start Event*, the central use case node, i.e. the service model determined within the node, becomes active. After execution, the central use case node triggers an asynchronous event for *Stop Event* that is initiated immediately. *Stop Event* is also linked to the second event control module of atomic mission models called *Stop Trigger*. When triggered, this module creates a new control flow token that is put on the output port *OUT*. Six global parameters were added for documentation purposes and requirements tracing. “*Associated_System_Name*” and “*Associated_System_ID*” are used to link atomic mission models to a superordinate mission model in association with parameters “*Associated_Mission_Label*” and “*Associated_Mission_ID*”. “*Atomic_Mission_Label*” and “*Atomic_Mission_ID*” are used to identify the respective atomic mission model.

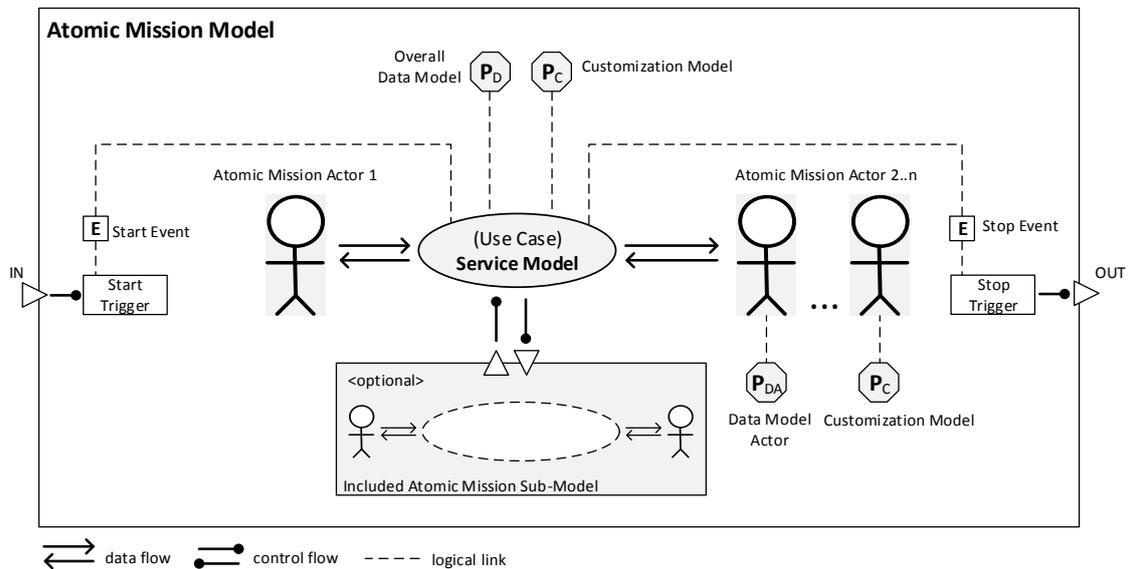


Figure 4.18 – Basic structure of an atomic mission model

Based on the definition for actors within the UML and SysML, actors (ACT) of atomic mission models can represent a wide variety of human and non-human actors that are associated with a use case. For instance, actors can represent users of the SuD, sensors or actuators of the SuD, external systems or environmental influences. Thus, actors can be internal to the SuD or external, i.e. they represent system context. In general, actors are responsible for triggering a use case or for receiving results [265]. For additional information on different types of actors please refer to UML and SysML literature e.g. references [265] and [365]. Actors of atomic mission models provide different parameters. As indicated in Figure 4.18, each general actor contains a parameter P_{DA} that stores an actor specific data model. Actor specific data models are linked to a global model parameter p_D that unites all actor specific data models in the form of a hierarchical data structure that is specific to each atomic mission model: $\forall P_{DA} \in \{Actor1..n\} : P_{DA} \subset P_D, P_D \in P_{AMIS}$. Special

customization actors do not contain an actor specific data model but a link to a global model parameter P_C ($P_C \in P_{AMIS}$) that contains a use case-specific customization model. Actors in the context of atomic mission models are described more closely within the following sub-chapter, *Actor Interaction Model Development*.

As described before, the central component of each atomic mission model is a specific use case of the SuD. This use case node UC contains a unique service model that is created as part of the systems engineering process in order to determine more specific functional and non-functional requirements for each service that is to be supplied by a SuD. The development of atomic mission models as well as service models is based on mission level and scenario-driven development approaches (cf. chapter 2.3.4.2 and chapter 2.3.4.1). Service models describe sets of different scenarios for each use case whereby scenarios basically describe sets of interactions between different actors of a system. Service model development is described in greater detail within chapter 4.3.5. A use case node is connected bidirectionally via ports and edges to each actor of an atomic mission model. These links are used to exchange data between service model and actors. Moreover, use case nodes are linked to two global parameters of the atomic mission model, P_C and P_D ($P_C, P_D \in P_{AMIS}$). As described before, parameter P_D contains a use case-specific data model. Parameter P_C contains a use case-specific customization model, in the form of hierarchical data structures. This model is used to determine different options for functional and non-functional design characteristics for each use case, e.g. with regard to different customers.

Alternatively, use case nodes can also be coupled bidirectionally to other atomic mission models. This is used for include relations, e.g. for use cases that require the integration of another basic atomic mission model. In that case, use case node and included atomic mission model do not exchange data but control flow tokens. In order to change control, a token is sent from a use case node to the associated atomic mission submodel. When the simulation of the submodel has finished, a control token is sent back to the original use case node.

4.3.2.1 Actor, Interaction and Data Model Development

During concept design, system context as well as stakeholders were determined. Moreover, different services, i.e. use cases, have been determined together with ideas for a top-level system architecture. In the course of atomic mission model development, the level of detail is increased for each use case. As part of this work, actors as well as system interaction points need to be determined for each specific use case, i.e. the central node of an atomic mission model. In the context of this work, the following definition shall be used for actors on top of the definitions provided by the UML and SysML:

Definition 11 *Actors are entities that interact with the boundary of a system under design in a way to perform or enable a specific use case. As part of each use case, actors intend to achieve certain goals or help achieving a specific goal for another actor.*

Actors are part of atomic mission models as well as overall executable specification models and are used to link both types of models. Moreover, actors trigger actions,

expect and receive results according to the service model described within the central use case node. It is often easy to determine human actors as part of a use case that are users of a SuD. In the case of non-human actors however, the process of actor identification may become more difficult, e.g. for actors that represent system environment inputs like temperature or noise. Non-human actors often contribute to achieving a goal for human actors, but there may also be cases with only non-human actors involved. However, it is important to think about the real beneficiaries of a use case that may also be indirectly related or passive human actors. If a service of a SuD is only provided to non-human entities, it may be considered not important for the SuD after all, since no one will miss the service provided. This decision needs to be taken with care for each use case by involving stakeholders as well. As part of a use case, actors interact with the SuD as well with its system environment via interaction points. In this context, one needs to differentiate between the terms interface and interaction point although both are similar.

Definition 12 *Interaction points* represent the boundaries of a system under design that are used to exchange information between actors and entities within the system as well as between actors and entities of the system environment.

In other words, interaction points determine links between actors and the overall system that are needed for any actor-system-environment interaction (cf. *validation points*, [8] and [47] and chapter 2.3.5). Thus, they are intended for use case related information exchange. This definition is more general than the definitions for interfaces in computing or human machine interface (HMI) design, since no definite technical solution is implied by an interaction point. Thus, no specific combination of hardware and software needs to be provided at this point of design for any interaction point, e.g. by defining a specific network interface. In order to exchange information between actors and interaction points within a formal simulation model, a data model needs to be created that describes structure and characteristics of the information to be exchanged. In contrast to actors, who represent entities of the real world, interaction points are abstract and imaginary constructs for actor to system interfacing that are not directly modeled as part of a use case but indirectly in the form of actor modules with specific data models. As system design and system development progresses, interaction points will be modeled as part of executable overall system specifications and, as a result of this process, will eventually be substituted by design solution-specific interfaces. At this point of design, system architecture information from previous development projects is used in order to determine actors and possible interaction points more closely. If not available, an abstract concept for overall system architecture can be developed as part of the conceptual design phase in order to provide a draft for the intended system.

In the context of this work, any use case related interaction between actors, system and system environment trigger state changes at interaction points that can be evaluated as part of a model execution, i.e. simulation. Thus, interaction points are observable. State changes at interaction points are communicated to associated actors or can be actively retrieved by associated actors of a model. As part of this process, communication is performed based on the data models that have been determined for each atomic mission model. States that can be evaluated for each interaction point include logical, physical as well as mixed characteristics. For

instance, a logical state may indicate a certain status of the SuD, e.g. “*system in maintenance mode*”, or of an HMI component, e.g. “*visual indication active*”. Physical characteristics include all kinds of quantifiable and qualifiable object states, e.g. “*temperature*”, “*power consumption*” or “*audio volume*”. The latter are often used for evaluating non-functional system requirements.

Figure 4.19 depicts the relation between atomic mission model, system under design and system environment. SuD and system environment interact with each other and exchange information, i.e. data as part of an executable model. However, from an atomic mission model point of view, system under design and system environment are treated like black boxes. This is because atomic mission models determine services and qualities that need to be provided by a SuD for a set of actors rather than system implementation specifics. Actors within atomic mission models are associated with system interaction points I that are determined during the creation of an atomic mission model.

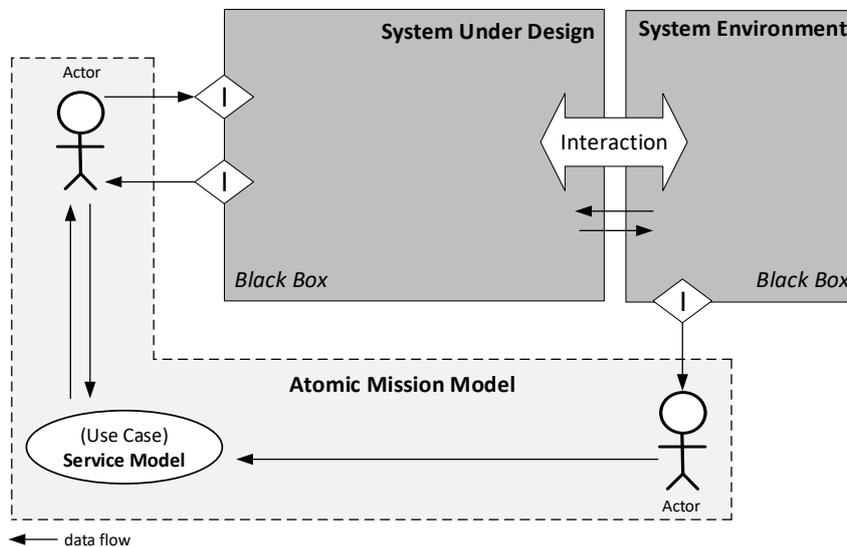


Figure 4.19 – Relation between atomic mission model, system under design and system environment (context) during development of an executable requirements specification. Actors are associated with specific interaction points I

At this point of design, system designers begin to develop a formal data model that determines the kind of information to be exchanged between actors and interaction points of a SuD. However, the flow of data exchange required in the course of atomic mission execution is determined in detail during the development of a concrete service model for each atomic mission model. Necessary changes that emerge from this phase of development are used to update the overall data model. A data model is derived from the intended use case of an atomic mission model as well as associated actors and possible interaction points. All information that needs to be exchanged between different entities is formalized in the form of data structure definitions. The conceptual design that has been described within an overall mind map is used as input for this point of development. This mind map not only includes requirements for essential use cases for the SuD that are realized in the form of atomic mission models but also preliminary data model descriptions, e.g. in the form of class diagrams or entity relationship diagrams (cf. chapter 4.2.2).

The design of a data model in the context of atomic mission model development can be divided into three sequential parts with associated questions and actions:

1. What actors are associated with the central use case described within an atomic mission model? (determine actors)
2. Where do these actors interact with the system under design or its environment? (determine abstract interaction points)
3. What information is to be exchanged between actors and interaction points in the course of a use case and what is the corresponding data model? (determine data model for each actor)

Since details of a use case are described later during the development of a service model for each atomic mission model, the overall data model of each atomic mission model needs to be adapted. In the course of this process, the last of the three questions described above will be used iteratively. Within atomic mission models, a global parameter $P_D \in P_{AMIS}$ exists that describes an atomic mission-specific overall data model. This data model is composed hierarchically of all data submodels for each actor associated to the central use case node, i.e. service model. For each atomic mission model, P_D points to a hierarchically structured data model that is integrated within an overall data model, which is composed of basic and use-defined data structures. Figure 4.20 illustrates the general structure of data models as part of atomic mission models in the context of the overall data structure model.

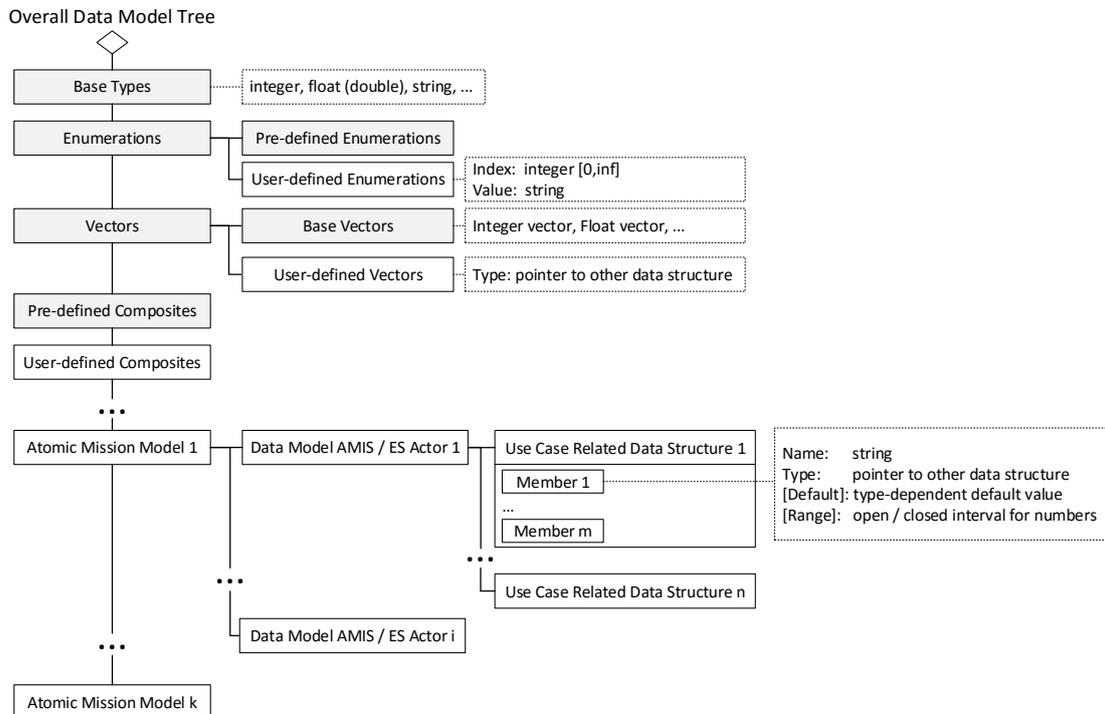


Figure 4.20 – Basic data structures (grey boxes), custom data structures and hierarchical data model structures for atomic mission models (white boxes)

Each data model that is created is stored as part of the overall atomic mission model. However, it is also possible to associate data structures created at atomic

mission level with higher level models, e.g. mission models. Top level node of each data model is the associated atomic mission model. One level below, data models for each actor are listed. Each of these actors contains a non-empty set of use case related composite data structures. Use case related composite data structures contain a non-empty set of different members. A member can be described by a 4-tuple (*name*, *type*, *default*, *range*) where:

- *name* = is a unique name to identify a member
- *type* = determines the data type of a member
- *default* = determines a default value
- *range* = determines a boundary for values (if applicable)

For each member, four groups of different types are available:

1. *Base Types*: used to determine basic data structure types including float, integer, list, pointer and string
2. *Composites*: used to determine composite data structure types
3. *Enumerations*: used to determine enumerated types, consisting of an ascending index with associated string type values
4. *Vectors*: used to determine array-like data structures. Vectors have a specific type, e.g. integer or composite. When used within a model, a specific length and default value needs to be specified.

Moreover, system designers may create additional data structures of type composite, enumeration or vector in order to refine higher level data structures, e.g. atomic mission model data structures. It is also possible to use the same basic data submodel for differently instantiated actors of a similar type. By this, the overall data model of an atomic mission model can be kept more general. Consider the example of a cabin management system. All cabin attendants can participate in a public address use case. By design, it is intended to provide pursers with higher priority when performing a public address than other cabin crew members. As a result, all cabin attendant actors have a priority parameter as part of their data model, which is instantiated according to their role, e.g. with highest priority for purser actors. Each data structure contains a unique name, library association, parent data structure association and description field. Changes within the data model of each atomic mission model that result from the development of service models during subsequent design steps are performed consistently since data model of atomic mission models and service models are linked via parameter P_D .

During later development stages an executable overall system specification (EOSS) is developed based on an executable requirements specification (ERS). This EOSS needs to provide all services and qualities that have been specified by atomic mission models of the ERS. Moreover, system interaction, data and customization models are directly inherited by the EOSS from the set of atomic mission models. After

the development of ERS and EOSS has been completed, an automated validation process is performed that requires to couple the execution of both models (cf. chapter 4.4.2). As part of this process, atomic mission models play a central role since they unite driver as well as evaluation model (cf. chapter 2.3.4.2). Driver and evaluation model are used to determine whether an EOSS is valid by ensuring that specific services and qualities are provided. The service model described inside the central use case node of an atomic mission model controls the execution of an atomic mission model by triggering actions, evaluating reactions and observing quality properties. Actors of atomic mission models are used for information exchange between ERS and EOSS during automated validation. Figure 4.21 depicts the relation between atomic mission models, as part of an ERS, and an EOSS.

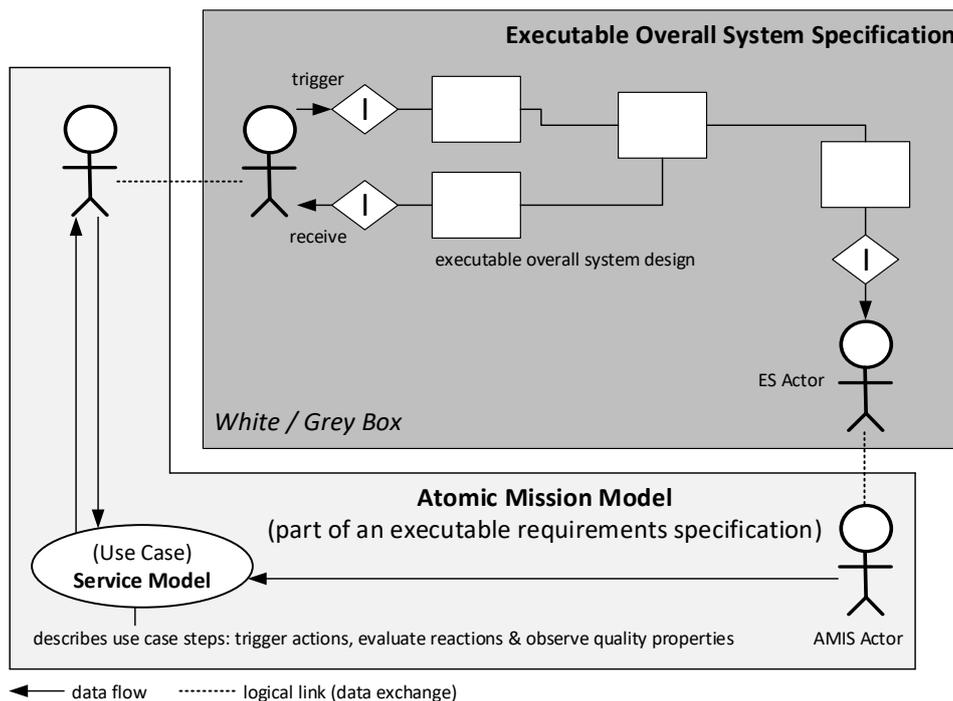


Figure 4.21 – Relation between actors of atomic mission models as part of an executable requirements specification and actors of an executable overall system specification

Both models are linked via two different types of actors, called *AMIS actors* and *ES actors*, in order to exchange information during execution of an automated validation process. As part of the atomic mission model, AMIS actors interact with the service model and provide a links to associated ES actors, which are part of the EOSS. On the other hand, ES actors interact with system architecture and system environment components of the EOSS, i.e. they are linked to system interaction points *I* that were determined during overall mission model development. Both types of actors complement each other and are developed during atomic mission model development. However, ES actors will be integrated within the EOSS later during development.

In the course of this work, one general as well as four task-specific AMIS actors with corresponding ES actors were developed for the associated design environment. Moreover, a second set of customization-specific AMIS and ES actors has been created. These actors represent a specialization of the model element actor and are described within the sub-chapter *Customization Model Development*. All developed

actors modules have been integrated within a dedicated library for the development of atomic mission models and can be used for modeling using a drag-and-drop mechanism.

Figure 4.22 depicts five different AMIS actor modules. On the left-hand side, a general actor module is depicted. This base module can be used to develop more specialized actor modules, e.g. in order to distinguish between different kinds of custom actors. In order to differentiate between actors of atomic mission models and actors of EOSS, the letters *M* and *ES* are used as part of each model component. All special actors that have been created as part of this work have a cubic shape. A plain cube shaped module, depicted on the right-hand side of the general actor module, is used for actors representing subsystems. On the right-hand side of the subsystem actor module, an actor module with sun pictogram is shown that is intended to be used for actors representing system context, i.e. environment. Right below the subsystem actor module, an actor for actuators is shown. This actor contains a gearwheel pictogram. The fourth special actor module, a sensor actor, is shown on the right-hand side of the actuator actor.

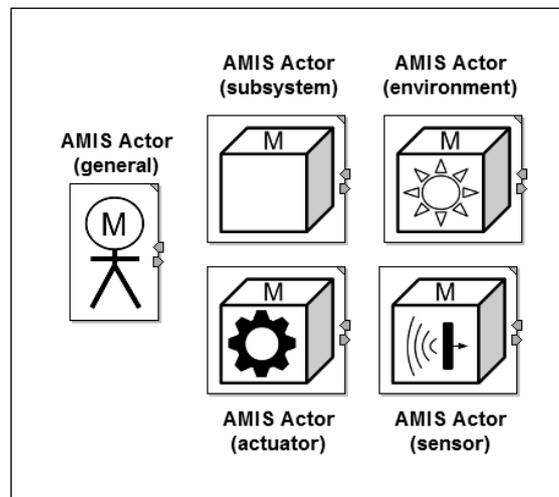


Figure 4.22 – Different actor modules for atomic mission models (AMIS)

Each AMIS actor has a set of six configurable parameters: *AMIS_Label*, *Name*, *Quantity*, *Role*, *Type*, *Actor_Data_Model*. Parameter *AMIS_Label* is linked to parameter *Associated_Mission_Label* of the associated atomic mission model. *Name* is used to determine a unique name for each actor of an atomic mission model. Moreover, this parameter is used together with *AMIS_Label* in order to link AMIS and ES actors during simulation as well as for documentation purposes. The third parameter that is important for linking AMIS and ES actors is *Quantity*. On one hand, *Quantity* determines if an actor represents a single actor (*Quantity* = 1) or a group of actors (*Quantity* > 1). On the other hand, *Quantity* is used in the form of an ascending progression of natural numbers in combination with parameter *Name* in order to address specific ES actors within the EOSS. Parameter *Role* is used to describe the role of an actor within the overall system and for documentation purposes. *Role* can be set to active, passive or mixed. Active participation means, that an actor represents an entity with specific goal that triggers a use case. A passive actor is recipient of use case related actions while a mixed actor shares both properties. Parameter *Type* is used to determine if an actor is a human, subsystem,

component or environment entity. Special actors have predefined roles that cannot be changed. Parameter *Actor_Data_Model* stores the data model of the respective AMIS actor.

In parallel to the definition of AMIS actors, counterpart ES actors are determined that will be integrated into an EOSS by system designers during later design stages. This is done by using pre-defined ES actor modules. Figure 4.23 depicts six different ES actor modules that were developed in the context of this work.

Two general actor modules were developed for representing single actor entities (bottom left) as well multiple actor entities, i.e. groups (top left). As with AMIS actors, four special actors have been created that are characterized by a basic cube shape with additional pictograms. These actors were derived from the general single actor module, but could also be transferred into multi-actor modules. A plain cube shaped module, depicted on the right-hand side of the general multi-actor module, is used for ES actors at interaction points representing subsystems. On the right-hand side of the subsystem actor module, an actor module with sun pictogram is shown that is intended to be used for actors representing interaction points with system context, i.e. environment. Right below the subsystem actor module, an actor with gearwheel pictogram is shown, intended for actuator interaction points. The fourth special actor module, a sensor interaction point actor, is shown on the right-hand side of the actuator actor.

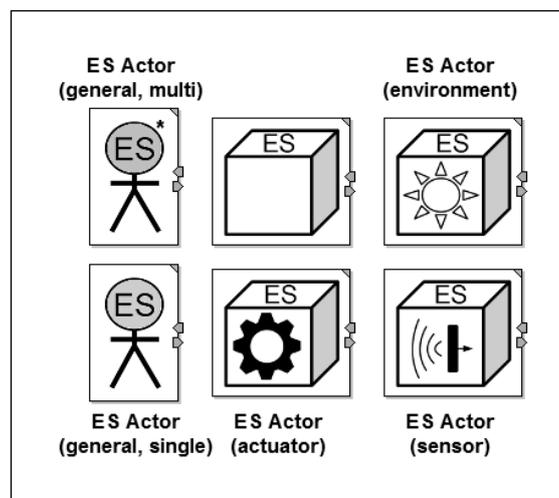


Figure 4.23 – Different actor modules for executable overall system specification (EOSS) models

Each single ES actor has a set of six configurable parameters: *AMIS_Label*, *Name*, *ID*, *Role*, *Type*, *Actor_Data_Model*. Except for parameter *ID*, all other parameters perform the same function than the parameters of AMIS actors. As described before, each AMIS actor contains a parameter *Quantity* that represents an ascending progression of natural numbers. Single type ES actors use parameter *ID* to determine one specific number of the set of numbers determined by parameter *Quantity*. For instance, if an AMIS actor exists with parameters *AMIS_Label* = “aLabel”, *Name* = “aName” and *Quantity* = 4, four single ES actor modules are created with parameters *AMIS_Label* = “aLabel”, *Name* = “aName” and IDs = 1 to 4 respectively.

Multi ES actors represent aggregated ES actors of the same type that are used to form groups of actors that may be scattered within the overall system model. In

contrast to single actors, multi ES actors have a set of seven configurable parameters: *AMIS_Label*, *Name*, *Continuous_ID_Range*, *ID_Enumeration*, *Role*, *Type*, *Actor_Data_Model*. All parameters except *Continuous_ID_Range* and *ID_Enumeration* perform the same function than the parameters of AMIS actors. Only one of both parameters can be set for each module. Parameter *Continuous_ID_Range* is used to represent a continuous group of ES actors with ascending IDs = [1, .., n] where $n \leq |Quantity|$ of the respective AMIS actor. On the other hand, parameter *ID_Enumeration* is used to represent a group of ES actors with different IDs in the form of an enumeration. Different IDs are separated by commas in order to form a list of actors of the same type: $ID_Enumeration = \{ID_1, \dots, ID_n\}$ where $|ID_Enumeration| \leq |Quantity|$ of the respective AMIS actor. AMIS and ES actors also provide the ability to choose a task-specific module icon and annotation parameter. This parameter can be used to determine a description for each actor as well as to describe associated interaction points more closely.

AMIS actors in combination with ES actors are crucial for information exchange between ERS and EOSS during automated validation. As part of this process, AMIS actors relay data between the service model and ES actors that are connected to specific system interaction points. At simulation start-up, initialization processes and model checking are performed by each AMIS actor. During this check-up, each AMIS actor determines if complementary ES actors are present with the EOSS, if the data model was initialized correctly and if all links have been established correctly. Logical links for data exchange are established by automatic creation of a global assignment table during simulation start-up. Figure 4.24 shows the linking mechanism between AMIS and ES actors. During initialization phase of each simulation, unique IDs are determined for each actor.

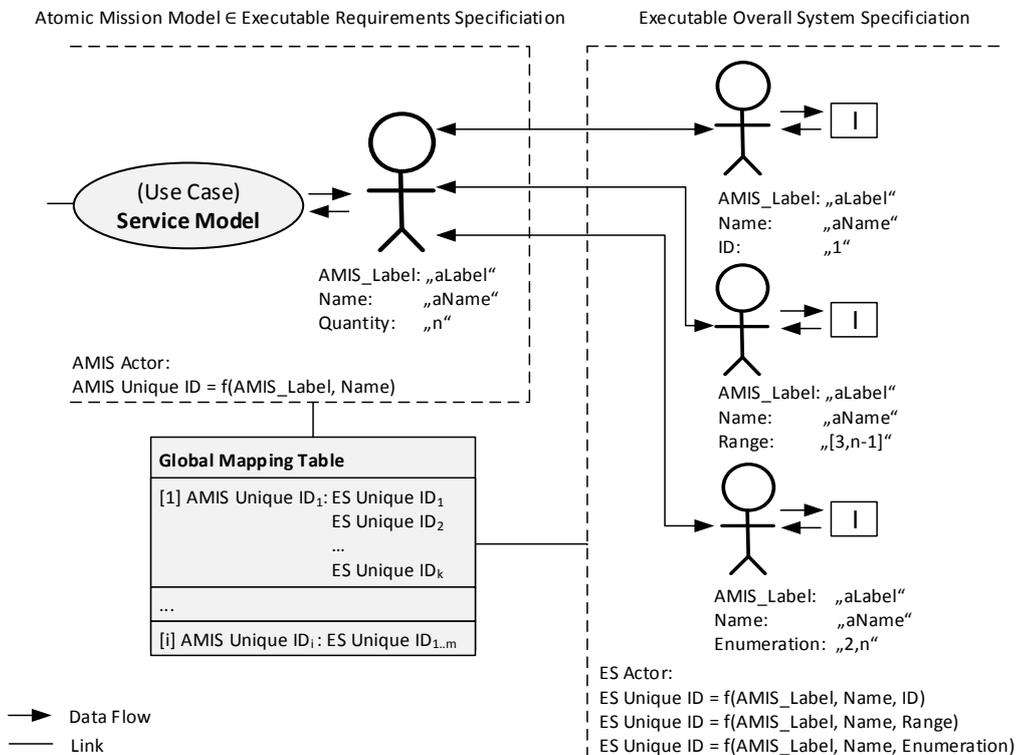


Figure 4.24 – Link mechanism between AMIS actors connected to a service model and ES actors connected to different interaction points I

Unique IDs for AMIS actors are calculated based on the content of parameters *AMIS_Label* and *Name* while unique IDs for ES actors are calculated in three different ways. Firstly, by using *AMIS_Label*, *Name* and *ID* parameters of ES actors representing a single actor. Secondly, by using *AMIS_Label*, *Name* and *Continuous_ID_Range* parameters or *AMIS_Label*, *Name* and *ID_Enumeration* parameters for actors representing groups of actors. When building a global assignment table, each AMIS actor is assigned a set of associated ES actors, which resembles a 1:n relation. During simulation, data flow between all actors is performed based on the mappings within the global assignment table. In consequence, each AMIS actor is able to relay data to a non-empty set of associated ES actors and vice versa.

During simulation of atomic mission models, messages are exchanged between AMIS and ES actors. Messages are specific instances of data structures of the overall data model. They are created by service models and instances of the EOSS. In order to relay messages between specific AMIS and ES actors, i.e. to address recipients for data particles, a predefined data structure called *ACTOR_DS* is used. Figure 4.25 depicts the general structure and characteristics of *ACTOR_DS*. Members *TYPE* and *ROLE* are set to the corresponding value of the respective AMIS actor module. *MESSAGE* is used to store the actual message that is to be exchanged between one or more specific actors (data encapsulation). *AMIS_Label* and *ACTOR_NAME* are used by ES actors to determine the recipient of a message. Messages received by ES actors from the EOSS model are encapsulated within a data particle of type *ACTOR_DS* and transferred to the associated AMIS actor with the aid of the global assignment table. On the other hand, messages received by ES actors from AMIS actors are unpacked and forwarded to the corresponding interaction point.

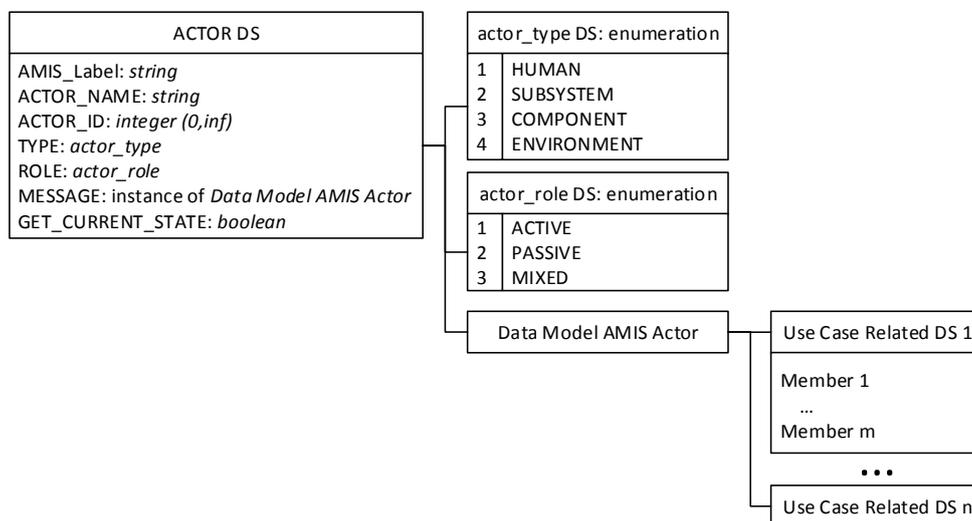


Figure 4.25 – Composite data structure *ACTOR_DS* is used to exchange data particles between AMIS and ES actors during simulation

In contrast, AMIS actors use *AMIS_Label*, *ACTOR_NAME* and *ACTOR_ID* to determine a specific recipient of a service model message within an EOSS. However, specific ES actor addressing and data encapsulation is managed directly by the service model embedded within each atomic mission model. If set to “*TRUE*”, member *GET_CURRENT_STATE* is used by service models with associated AMIS actors

to request the current state of an ES actor, i.e. the corresponding interaction point. In this case, an automatic response is sent by the respective ES actor, containing the last message that was received by the ES actor from the associated interaction point of EOSS model.

In the course of simulation, an automatic type check is performed based on the data model determined by parameter *Actor_Data_Model* for each message that is exchanged between AMIS and ES actors. This is done in order to verify that exchanged messages are compliant with the overall data model. In the case of using non-compliant data structures during simulation, a detailed error message will be displayed in the simulation log and automated validation will fail. It is important to note that AMIS as well as ES actors are only used for modeling data flow. Control flow during atomic mission model execution is modeled as part of the central use case node, i.e. the service model. In addition to the ability to relay information between EOSS models and atomic mission models that are part of an ERS, ES actors also store the current information or state that is available at a specific interaction point. With this, the current state of each interaction point is available throughout the overall simulation process, e.g. during automated specification validation.

During later development stages, e.g. when building an EOSS, a data processing module may be placed between ES actors and associated system architecture components. This is done in order to translate abstract atomic mission model data into more specific data of the overall system model, e.g. in order to account for specific input / output requirements of a re-used system architecture component. In this case, system designers need to ensure overall data model consistency. Moreover, specific parameters and parameter configurations of architecture model components, e.g. HMIs, can be derived from AMIS and ES actor parameter configurations as well as from associated data models.

4.3.2.2 Customization Model Development

In addition to the development of an interaction and data model, a use case-specific customization model is developed based on conceptual design. Customization plays an important role during system design, e.g. in order to provide different customers with possibilities for optional or unique system characteristics. Although design alternatives are evaluated during concept design, a more specific and formal determination of customization alternatives is required for automated design validation as well as in order to determine a finite design space of the SuD. Less confined customization possibilities often lead to a large number of possible parameter permutations that do not necessarily lead to useful or feasible design solutions. Customization also plays an important role for the development of a human machine interface concept model (cf. chapter 4.3.6).

As part of the development of atomic mission models, customization is expressed by sets of parameters that are determined based on the intention of the central use case node. Thus, a customization model is composed of sets of design parameters. As described before, this process requires to review the data model that has been determined before, in order to account for necessary changes that arise from customization. Moreover, it may also be necessary to create additional actor types. Customization models as well as data models will be refined further during implementation of concrete service models for each atomic mission model (cf. chapter 4.3.5).

All customization parameters that are determined during atomic mission model development are aggregated within a global model parameter $P_C \in P_{AMIS}$ in the form of a data structure model. This is done similar to the development of an overall data model for each atomic mission model. Figure 4.26 depicts the hierarchical structure for customization data models as part of atomic mission models. In contrast to the data model of an atomic mission model that is composed of data models of each AMIS actor and stored in parameter $P_D \in P_{AMIS}$, the customization model is exclusively stored in parameter P_C . AMIS or ES actors of atomic mission models do not have a link to P_C . A special customization actor type that does contain a link to data structures of parameter P_C is described later within this section.

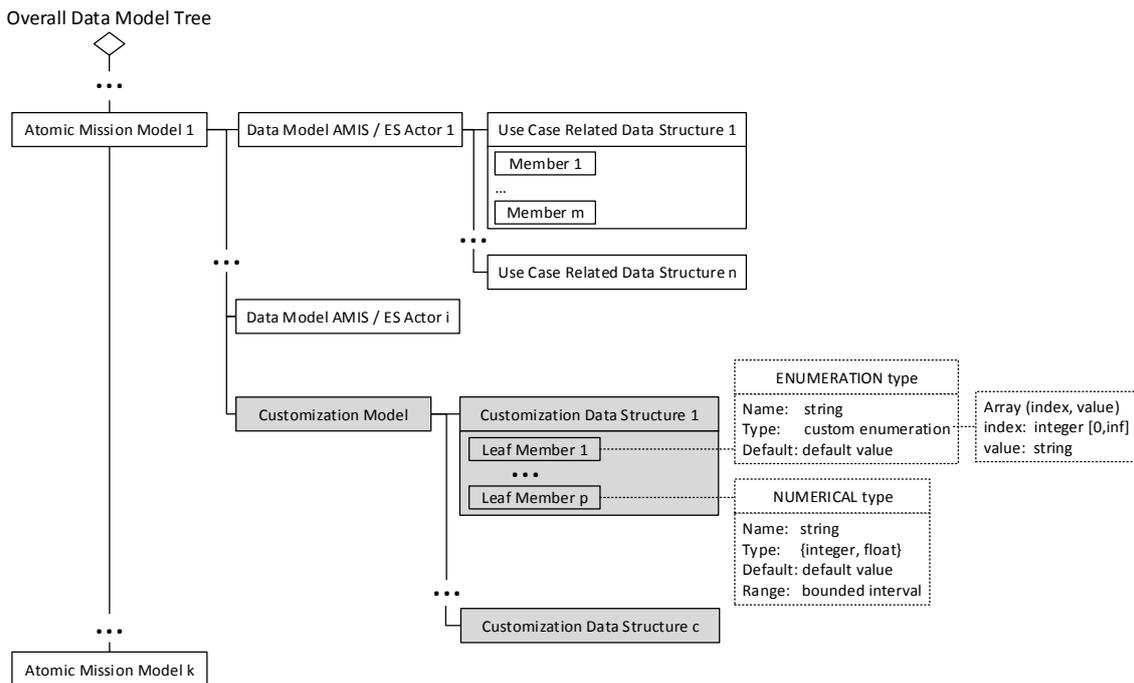


Figure 4.26 – Hierarchical customization data model structure (grey boxes) of atomic mission models with enumerated and bounded numeric data types (extension of Figure 4.20)

Parameter instantiation is performed by each service model during overall system simulation. In contrast to data models, customization models only use composite data structures, enumerated types and numerical data types with bounded interval. Composites are used for modeling hierarchical structures while leaf members of customization models can only be of type enumeration or numerical data types with a finite range, determined by minimum and maximum values. This is done in order to limit the design space for customization by providing finite sets of possible parameter configurations instead of arbitrary settable values, e.g. parameters that represent floating point values with infinite range.

In the context of this work, an enumerated type is defined to be a finite collection of $n \in N$ string type objects that can be indexed (array): $Enum = \{(i_1, v_1), \dots, (i_n, v_n)\}$, where $i \in N$ is an index number in ascending order (1..n) and v is a value of type string. With this, data type Boolean (true, false) can be mapped to an indexed enumerated type as well finite numeric lists that contain numbers of integer or float types.

During the development of atomic mission models, parameter customization can affect functional as well as non-functional design properties. The leading questions for the development of a customization model in the context of atomic mission models can only be answered by combining conceptual design, stakeholder knowledge and experience. Most significant stakeholders for this process are customers of the SuD (try to increase the overall number of design alternatives) as well as system providers (try to limit design parameter alternatives). In the course of customization model development, three general questions need to be answered:

1. What parameters of an atomic mission model, i.e. use case, are generally applicable for all stakeholders and cannot be customized? (determine fix parameters, part of data model development)
2. What parameters of an atomic mission model, i.e. use case, can be chosen individually by all stakeholders? (determine variable / optional parameters)
3. What specific characteristics do variable and optional parameters have? (determine variable and optional parameter characteristics)

Parameters can be created to express sets of functional and non-functional characteristics, conditions and constraints as well as actor assignments. Fix design parameters that are related to a use case of an atomic mission model are typically already determined during concept design and actor data model development. They are not used for customization since they are mandatory part of the overall design.

Architecture customization parameters are used to extend the non-functional or quality properties related to a specific use case. Thus, they represent configurable system architecture properties for the SuD, i.e. customers are free to choose parameter values from the set of predefined values for manufacturing or may accept the default values provided.

Functional customization parameters are used to extend or limit the functional possibilities of a use case. In addition, customization parameters can also be used to activate or deactivate a specific service that is provided as part of a use case. In this case, the respective atomic mission model use case is considered an optional system service.

All variable customization parameters described so far can be categorized to be observable system attributes that can be assessed by actors at respective system interaction points. This is because they describe functional or quality-related service aspects that are expected by actors during utilization of a service. In general, actors expect certain qualities in the form of structured information that can be observed at an interaction point. Such an interaction point provides a specific functional and/or non-functional output when utilizing a specific service. As a result of the determination of variable customization parameters for functional and quality characteristics, actor data models may need to be updated. In the case of non-functional, i.e. quality customization parameters $NFCP \in p_C$, respective data structures $DS_{1..n} \in NFCP$ are directly inherited to the data model of affected actors $DMA_{1..i}$ as depicted in Figure 4.27.

For function relevant customization parameters $FCP \in p_C$, it may become necessary to determine new function characteristic data structures $DSF_{1..f}$ as part of the data model of affected actors $DMA_{1..j}$. Necessary data structures for AMIS and ES actors are created according to the overall intention of the respective use case in combination with the intention of specific customization parameters.

In case additional quality characteristic parameters $N\widehat{FCP}$ are associated with function relevant customization parameters FCP , respective data structures $DS_{1..q} \in N\widehat{FCP}$ are directly inherited to the data model of affected actors $DMA_{1..j}$ so that $DMA_{1..j} \cap N\widehat{FCP} = DSQ_{1..q}$. In some cases, as indicated in Figure 4.27, it may also become necessary to determine new actors with adequate function and quality related data models in order to account for functional or non-functional customization parameters. The development of new actors has to be performed in accordance with the overall intention of the use case of an atomic mission model. The principle of determining new actors, function-related data structures as well as inheriting customization qualities to actor data models is most important for the creation of service models as well as for the process of automated validation in order to evaluate a specific system design.

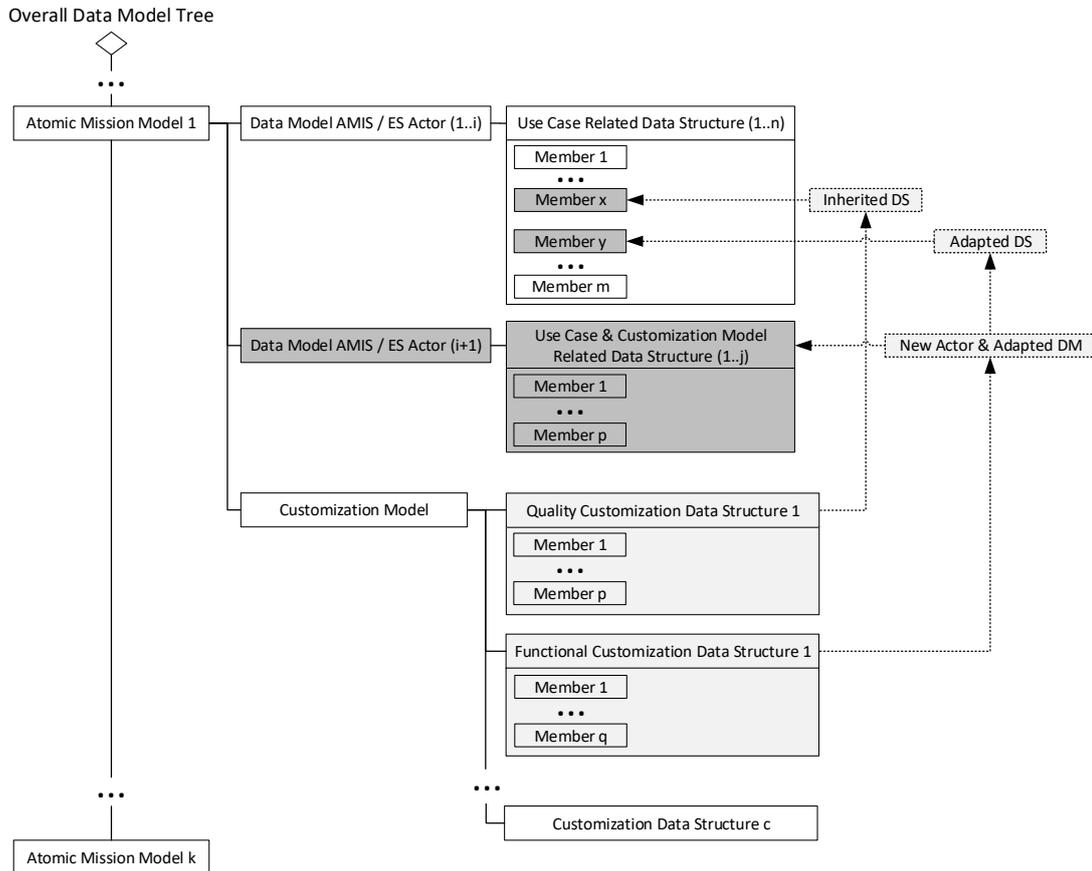


Figure 4.27 – Customization model development (gray boxes) may lead to extension of the overall data model (dark gray boxes). Quality related customization data structures DS are directly inherited to respective actor data models (Member x) while function related customization data structures lead to extended actor data models with adapted data structures (Member y) or yield new actors (dark gray) with adapted data model (DM)

In the case of function-related customization parameters, two special cases exist. Firstly, customization parameters that describe function-related conditions or con-

straints for events. Secondly, customization parameters that describe function-relevant actor assignments. Both types of parameters are expressed with logical expressions, i.e. Boolean algebra.

Conditions as well as constraints are used to determine sets of linked events that have a functional impact on a specific atomic mission model use case. In terms of atomic mission models, events can originate from (input events) or affect (output events) internal actors of the system under design as well external actors that are part of system environment. The intention behind this type of customization parameters is to provide a set of different customer-selectable conditions or constraints applicable for a specific service, e.g. in order to determine a set of distinct actor inputs that trigger a specific use case.

As part of customization models, conditions and constraints are expressed by using a truth-functional approach. In this case, customization parameters of enumerated type are used with values of type string, excluding blanks, that express atomic propositional logic formulae. These logical formulae only use variables, i.e. events, that can be evaluated to be true or false as well as the logical operators conjunction (AND), disjunction (OR) and negation (NOT) in combination with round brackets. In addition, the laws of Boolean algebra are applied in order to form expressions. See, for instance, reference [20] for more information on logic formulae and Boolean algebra. A customization parameter used to determine conditions or constraints for a specific use case is described by a non-empty data structure of enumerated type $CC_Enum = \{(i_1, v_1), \dots, (i_n, v_n)\}$ where $i \in N$ is an index number in ascending order (1..n), $|CC_Enum| > 1$ and each value v represents a logical customization expression LCE :

$$S_v = LCE$$

$$LCE := e | \neg LCE | LCE \wedge LCE | LCE \vee LCE | (LCE) \text{ where } e \in Actor_Events$$

$$Actor_Events := \{ActorInputEvents, ActorOutputEvents\}$$

$ActorInputEvents$ and $ActorOutputEvents$ can be described by a 2-tuple ($Identifier$, $Value$) where:

- $Identifier$ is logical variable name of type string (excluding blanks)
- $Value \in \{TRUE, FALSE\}$ (Boolean type)

Each $Identifier$ is created based on the following naming convention: $Actor:Event$ $Actor$ is the name of an atomic mission model actor and $Event$ is the name of the specific event (data structure) of type Boolean that is part of the data model of the respective actor.

Members of $Actor_Events$ represent sets of logical variables as well as data structures of the data model of associated atomic mission model actors. For each logical variable, only element $Identifier$ is part of logical customization expressions. If logical variables are set "TRUE", the respective event has occurred. Variables set to "FALSE" determine the absence or inactivity of a specific event. Specific evaluation of logical customization expressions in the context of use cases is determined later as part of service model development. In the context of this work, C/C++ programming language logical operators are used as part of logical customization expressions LCE , where $\neg := !$ (NOT), $\wedge := \&\&$ (AND), and $\vee := ||$ (OR).

Each customization parameter that describes a set of logical expressions CC_Enum is named with an appropriate name in order to indicate its purpose. Additional information can also be included within the description field of each data structure. For condition or constraint-relevant customization parameters $CCP \in p_C$, it may become necessary to determine new data structures $DSCC_{1..c}$ as part of the data model of affected actors $DMA_{1..a}$. As before in the case of functional customization parameters, it may also be necessary to determine new actor types with according data model in the course of developing condition or constraint customization parameters.

As an extension of condition and constraint-based customization parameters, function-relevant actor assignments are used to customize the relation between a use case and its actors. In other words, function-relevant assignments represent input and output assignments for actors and service models of atomic mission models. They determine actors that trigger specific actions of a use case (initiators) as well as actors that are targets of actions of a use case (recipients). Customization parameters used to determine function-relevant actor assignments for a specific use case are described by a non-empty data structure of enumerated type $FAA_Enum = \{(i_1, v_1), \dots, (i_n, v_n)\}$ where $i \in N$ is an index number in ascending order (1..n), $|FAA_Enum| > 1$ and each value v represents a logical actor assignment expression $LAAE$:

$$S_v = LAAE$$

$LAAE := a | \neg LAAE | LAAE \wedge LAAE | LAAE \vee LAAE | (LAAE)$ where $a \in Actors$

$Actors :=$ is a finite non-empty set of names of atomic mission model actors

Thus, function-relevant actor assignment parameters are basically lists of atomic mission model actors or sets of logically linked actors with appropriate name to indicate their meaning. As in the case of condition and constraint-based customization parameters, logical operators of the C/C++ programming language are used.

An assigned actor for inputs and outputs of a use case can be kept abstract in order to specify that either *actor A* or *actor B* can trigger a use case or be the target of specific use case actions. Alternatively, an already existing atomic mission model *actor A* can be split into more detailed versions *actor A₁..actor A_n*, e.g. because of already defined HMI concepts. In that case, actor splitting is used in order to differentiate between a set of different switches of one user interface that can be used by the same type of actor. However, all actors that can be assigned to participate in a use case need to have the same data model.

Customization and data model development are integral parts of atomic mission development and thus, the development of an executable requirements specification (ERS). Moreover, both models are crucial for the development of service models as well as related executable overall system specifications (EOSS). While atomic mission models determine data and customization models, service models and EOSS models use both implicitly. This means, that both, service models as well as EOSS provide different model properties for specific settings of customization parameters. Due to the range of possible settings for customization parameters, service model development needs to ensure, that all stakeholder-relevant customization possibilities and combinations have been determined and modeled with regard to the conceptual design.

When executing atomic mission models as part of an overall mission model, e.g. during automated validation of an EOSS, all models involved need to operate on the same set of customization parameters settings (customization model consistency). As a result, customization model settings of a specific atomic mission model need to be transferable during simulation. For data models, the bridge between ERS and EOSS is provided by using logically connected AMIS and ES actors. For customization models, a similar mechanism is provided. A pair of special actors is used in order to provide consistency between customization models of different superordinate executable models. Figure 4.28 depicts customization actor modules for atomic mission models (AMIS actor) and EOSS models (ES actor).

Both customization actor modules provide a set of three configurable parameters: *AMIS_Label*, *Name* and *Customization_DS*. *AMIS_Label* and *Name* are string type parameters that are used to determine a unique name for each actor. *Customization_DS* is a composite data structure type parameter that stores the specific customization model of an atomic mission model. Following the linking mechanism described for data model development earlier within this chapter, parameters *Name* and *AMIS_Label* are used to link AMIS and ES customization actors during simulation in order to exchange information, i.e. customization settings.

Customization model settings are sent from atomic mission models to EOSS models during overall model execution, e.g. during automated validation. As shown in Figure 4.28, AMIS customization actors provide an input port that is used to send customization information from service models, while ES customization actors provide an output port in order to relay information to EOSS. Customization parameter settings are transferred on the initiative of service models only.

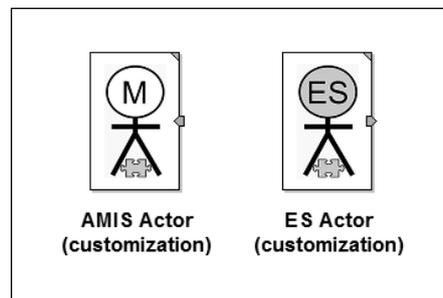


Figure 4.28 – Customization actor modules for atomic mission models (AMIS actor) and executable overall specification models (ES actor)

Figure 4.29 depicts the linking principle of customization actors based on a global assignment table for all atomic mission models that is automatically been built during simulation start-up. Again, unique IDs are calculated for each actor. For customization actors, unique IDs are based only on the label of the overall atomic mission model and the respective names for each actor. Actors with identical label and name are able to exchange information. In contrast to data model information exchange, customization model information exchange is one way only, from AMIS to ES actors. By design, possible relations between AMIS and ES customization actors include $1:1$ and $1:n$ cardinality. ES customization actors are connected to special interaction points of an EOSS that store customization model settings. As customization settings may not be stored centrally as part the concept for a given SuD, any number of ES actors with the same values for parameters *AMIS_Label* and

Name can be used in order to relay customization information at the same time to different entities of an EOSS. Figure 4.29 shows an example with two customization storage entities at EOSS side.

At simulation start-up, model checking is performed by each AMIS actor. As part of the initialization process, the default configuration of each customization model is transferred to associated ES actors to ensure consistency at simulation start-up. Moreover, each AMIS actor analyzes, if complementary ES actors are present with the overall system model, if the customization model was initialized correctly and if all links have been established.

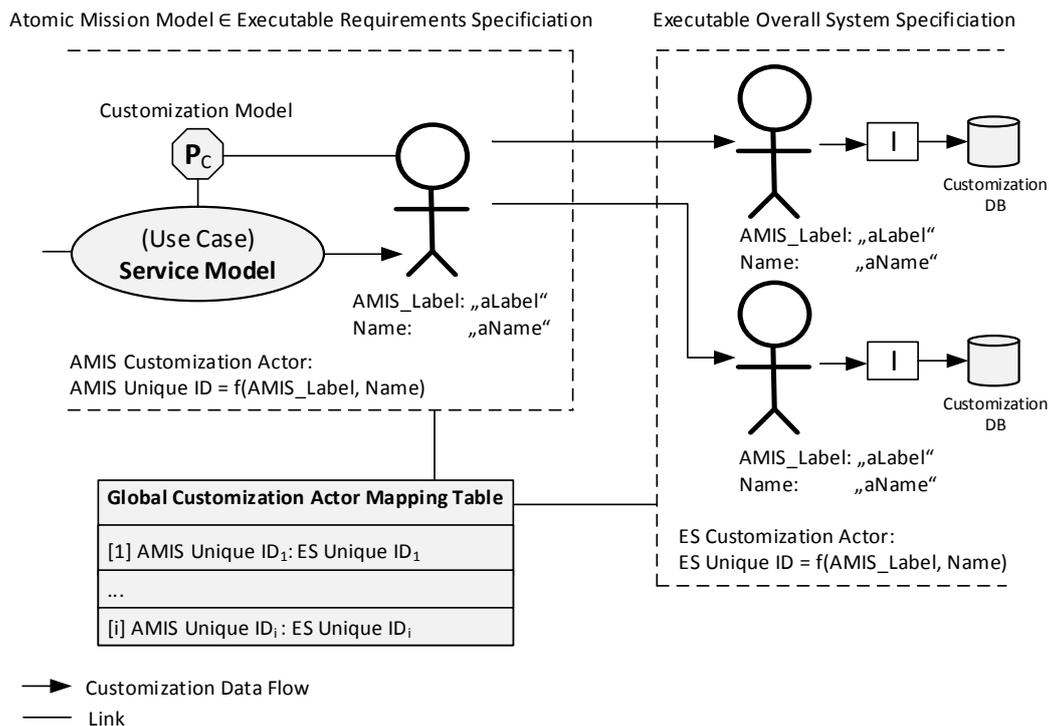


Figure 4.29 – Customization model exchange between an AMIS customization actor and two separate ES customization actors connected to different interaction points I that are connected to customization parameter databases (DB)

4.3.3 Quality Objective Model Development

Like atomic mission models, quality objective models (QOM) represent essential building blocks of an overall mission model. They determine non-functional requirements, i.e. quality requirements. In combination with atomic mission models, QOM are an important prerequisite for enabling automated validation of executable overall system specifications (EOSS). As part of this work, the following definition shall be used:

Definition 13 A *quality objective model* is a formalized discrete event module with associated parameter set. It is used as part of mission models to determine quantifiable non-functional requirements at system level. The characteristics of a non-functional requirement are determined with the aid of an objective parameter with bounded value range. Moreover, quality objective models are used for automated validation in conjunction with non-functional observer modules that are part of executable overall system specifications.

Requirements described by quality objective models are also referred to as global quality requirements. Global or system quality requirements include, but are not limited to, weight, cost, power consumption / power budget, overall installation space, heat dissipation, Mean Time Between Failures (MTBF), and Mean Time to Repair (MTTR). Quality requirements that are described as part of atomic mission and service models are also referred to as local or use case-related quality requirements. Modeling of combined functional and non-functional requirements for each atomic mission model in the context of service model development is described in chapter 4.3.5. Quality requirements are specified with the aid of objectives and are typically determined by system architects and designers in order to determine ranges or budgets for non-functional system parameters such as overall weight or power consumption. The following definition for objectives is based on the definition for weighted objectives by Marwedel et al. [215]:

Definition 14 An **objective** o is a 4-tuple (n, lb, val, ub) with objective name n of type string and real numbers lower bound lb , target value v and upper bound ub representing the characteristics of non-functional requirements at system level with $lb \leq val \leq ub$. Number v determines the target value or mean for a non-functional requirement. Numbers lb and ub determine a bounded parameter range to reflect the uncertainty of the corresponding non-functional design parameter. An objective o_i is said to be attainable by the system under design, if the corresponding observed value ov_i of a non-functional design aspect during execution of an executable system specification satisfies the following condition: $ov_i = \{ov_i \in R, |, lb \leq ov_i \leq ub\}$.

All objectives $o_1..o_n$ for a system under design are aggregated at mission level in the form of an overall objective set O . O is a composite data structure that is integrated within the overall data model tree of a mission model with n members. Each member is of type objective. During creation of each quality objective model, a second formalized building block called non-functional observer (NF) is created and parameterized in parallel. This module is linked to the associated quality objective model and is able to observe non-functional properties of an EOSS during simulation. A non-functional observer is defined as follows:

Definition 15 A **non-functional observer** is a formalized discrete event module that is used as part of executable overall system specifications. It represents a data probe that is coupled to a specific quality objective model in order to observe quantifiable non-functional parameters during simulation. Non-functional observers are created and parameterized in conjunction with quality objective model development.

Non-functional observer blocks are used by designers of an later EOSS models in terms of parameter evaluation probes. This means, that non-functional observer modules are linked to system architecture entities of the EOSS, including subsystems and *elements* (cf. chapter 4.4). In general, an *element* represents a system architecture component, e.g. a line replaceable unit (LRU).

The creation of quality objective models with associated non-functional observer modules can be compared to the creation of AMIS and ES actors during the development of atomic mission models (cf. data model development, chapter 4.3.2).

During automated validation, quality objective models will validate non-functional requirements by evaluating data received from their associated non-functional observer modules. Figure 4.30 depicts the relation between quality objective models with non-functional objective parameter (bottom) as part of an executable requirements specification and non-functional observer modules as part of an executable overall system specification (top).

In the course of this work, four predefined modules were developed for the associated design environment. These modules provide a drag-and-drop mechanism. When creating a quality objective model instance, designers specify a name for the associated non-functional requirement, a bounded objective and determine a label for the system under design. In parallel, a non-functional observer module instance is created that carries the same information. Both modules are automatically coupled and exchange information during overall system simulation. Information flow is one way only, from NF to QOM.

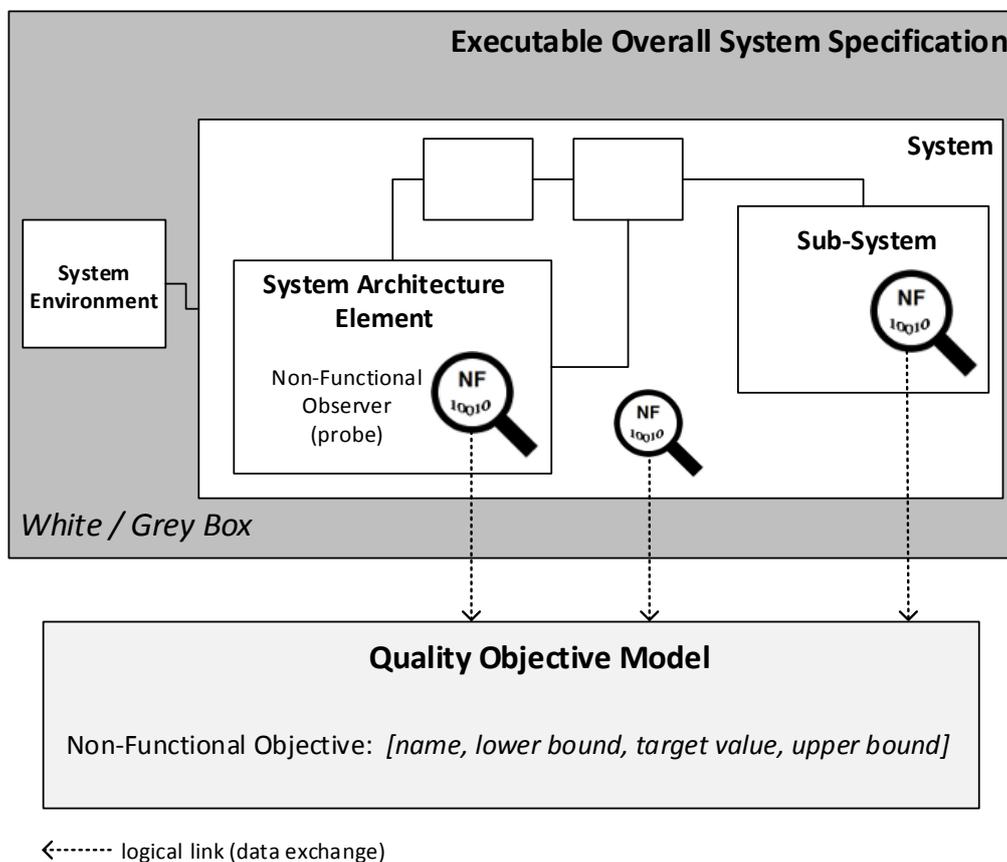


Figure 4.30 – Relation between quality objective models with non-functional objective parameter (bottom) as part of an executable requirements specification and non-functional observer modules (NF) as part of an executable overall system specification (top)

Developers of an executable overall system specification are able to place an arbitrary number of NF modules of the same type within the architecture model of the SuD as shown in Figure 4.30. NF modules can be placed at system level to observe global design parameters or inside other hierarchical architecture blocks, including subsystems and elements. They can directly be linked to parameters of the respective entity of an EOSS model or connected to specific interaction points that provide information on non-functional system properties.

Information on non-functional properties can be accessed, accumulated and evaluated by QOM modules at any time during overall system simulation. Each QOM module performs an automated check in order to determine if information was accessible during simulation. During automated validation, each quality objective model will automatically provide validation status information. This information is included within the overall validation report. More information on the process of automated validation is provided in chapter 4.4.2.

Two types of quality objective models exist for determining non-functional requirements at system level. Firstly, a quality objective model can be used to determine a non-functional requirement for a SuD that is not affected by system application. In the context of this work, this type of quality objective models is called static quality objective model. Quality requirements described by this type of model are not affected by the application of use cases of the SuD, i.e. the specifics of atomic mission models during execution. Typical non-functional requirements determined by static quality objective models are overall system cost, installation space or weight.

Secondly, a quality objective model can be used to determine a non-functional requirement for a SuD that may be affected during system application. In the context of this work, this type of quality objective model is called continuous quality objective model. Typical non-functional requirements determined by continuous quality objective models are power consumption or heat dissipation. The differentiation between both types of objective models for a given non-functional design aspect is not always identical for different systems and needs to be determined separately for every development project. When developing a cabin management system for instance, the non-functional requirement “*system weight*” can be considered static, since no material flow exists during operation that affects the weight of the system. However, at higher level of abstraction, at aircraft level, non-functional requirement “*system weight*” could also be considered part of a continuous quality objective model since an aircraft will lose weight during flight by burning fuel.

As part of this work, three different quality objective model modules were developed for the associated design environment. In addition, a pre-defined non-functional observer probe has been created that is used as part of executable specification models in the context of automated validation. Figure 4.31 depicts all four model components. Quality mission model component A1 is used to model detached or parallel mission model nodes. These nodes are not connected to the control flow of a mission model and start execution in parallel with the associated overall mission model. Execution ends with the conclusion of the overall mission model. This type of module is used in order to determine and validate continuous quality requirements that are persistent during overall system application, e.g. to determine a range for overall power consumption.

A similar module, A2, can also be used to determine continuous quality requirements. In this case, a control flow input port is provided that allows to connect this type of module to the overall control flow of a mission model. Thus, module type A2 starts execution only after receiving a control flow token at a specific point during overall mission model execution. As before, execution of this type of continuous quality model ends with the completion of the associated overall mission model. This specific type of quality objective model is used to determine and validate continuous quality requirements that are not applicable during the entire process of

overall system application. In contrast, it is used to determine quality requirements that are applicable beginning at a specific point during a mission and persist until the end of a mission. For instance, in order to determine quality requirements that require a specific set of preconditions.

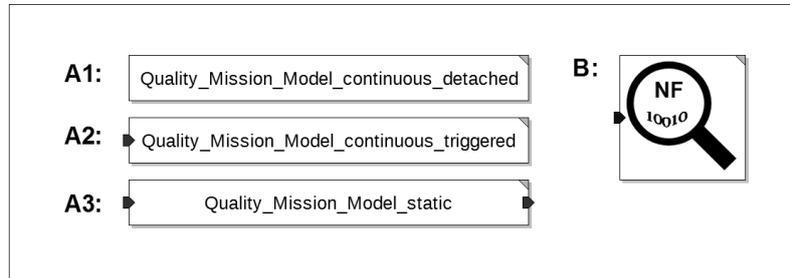


Figure 4.31 – Quality objective model modules for mission models (A1, A2, A3) and non-functional observer probe (B) for executable overall specification models

Consider an example with a continuous quality requirement for overall power consumption of a cabin management system during flight. This requirement shall only be applicable for power that is generated by the power plant or auxiliary power unit (APU) of an aircraft. When on ground and connected to ground power, this requirement may be applicable in a different form or not at all. Thus, a mission model containing this specific quality model will use module type A2 after a specific point of the overall mission model where the aircraft is considered to be disconnected from ground power provision, e.g. when in flight.

Module A3 shown in Figure 4.31 is used to determine and validate static quality requirements as part of a mission model, e.g. overall system weight. Similar to atomic mission models, this type of module provides an input and output port for control flow tokens. It is thus used at a specific position within a mission model. It is executed when triggered and produces a control flow token output when finished. Quality objective models monitor, accumulate and update information on non-functional design aspects by accessing associated non-functional observer probes during simulation. Both types of modules operate event-based.

Each of the quality objective model modules A1 to A3 depicted in in Figure 4.31 has the following set of configurable parameters. Parameters *Associated System Name* and *Associated System ID* are used to determine the associated system under design. *Associated Mission Label* and *Associated Mission ID* are used to link quality objective models to a superordinate mission model whereas *Non-functional Parameter Name*: determines a specific name for the non-functional system requirement. Parameter *Objective* [n, lb, val, ub] determines an objective for a non-functional system requirement.

As described before, the non-functional observer probe (B) depicted in Figure 4.31 is used by developers of executable overall system specifications. It is used to probe and monitor executable system specifications in order to provide information regarding non-functional design aspects to associated quality objective models during automated validation. Six configurable parameters exist in total. The first three of these parameters are determined by designers of the respective quality objective model [QOM] whereas the last three are configured later during executable specification development [ES]. Parameter *MissionLabel* [QOM] determines the label

of the associated quality objective model while *SystemName* [QOM] determines the associated system under design. *Non-functional Parameter Name* [QOM] determines the name for the associated non-functional system requirement. On the other hand, parameter *StaticParameterValue* [ES] is used to determine a static value for a non-functional design aspect. This parameter is settable or can directly be linked to an appropriate design parameter of an related EOSS model. *ElementName* [ES] is used to determine the name of a specific system architecture entity that is observed by the respective non-functional observer probe (overall system, subsystem, element). This parameter is settable or can directly be linked to an appropriate design parameter of an EOSS model. The optional parameter *ElementID* [ES] can be used to determine an identification number for a specific system architecture entity that is observed by the respective non-functional observer probe (overall system, subsystem, element). Again, this parameter is settable or can directly be linked to an appropriate design parameter of an EOSS model.

4.3.4 System and Environment Configuration Model Development

As described in the first sections of chapter 4.3.1, configuration of system as well as system environment parameters play an important role in the creation of mission models. According to Liebezeit [198], the configuration of a system represents its initial state or its current state at a specific point of time in the course of mission execution. The same is applicable to system environment configuration [198]. As part of this work, system and system environment configurations are used to shape the global flow of mission models in a way to enable specific conditions and prerequisites for embedding and executing sub-mission models, atomic mission models (AMIS) and quality objective models (QOM). Since an executable overall system specification (EOSS) unites system under design (SuD) and system environment, both types of configuration need to be considered during the development of mission models.

Definition 16 *A system and system environment configuration model is a 2-tuple (S_C, E_C) that determines sets of configurable system parameters S_C and system environment parameters E_C with associated characteristics at overall system level for a system under design. Both sets are independent from each other and represent finite lists of configuration parameters $\{cp_1, \dots, cp_n\}$, $n \in N$. A configuration parameter is an object with variable degree of complexity and can be described by a 4-tuple (name, type, default, range) where name is a unique identifier for a given parameter, type either determines a basic data type, an enumerated type or a composite data structure, default determines a default value (e.g. to describe initial parameter states) and range determines bounded value ranges, if applicable (only used for numerical basic data types).*

System as well as system environment configuration parameter sets S_C and E_C are part of the overall data model tree of a mission model and are placed at the same hierarchical level then atomic mission models. Both sets are represented by composite data structures with a number of $|S_C|$ and $|E_C|$ members respectively. Moreover, S_C is part of the overall parameter set of the system model SM of an EOSS whereas E_C is divided into n subsets $E_{C1} \dots E_{Cn}$ for each system environment model $SEM_1 \dots SEM_n$ of an EOSS.

By changing parameter values for system and system environment configuration parameters, different conditions and states can be stimulated for a SuD during simulation, i.e. automated validation. System and system environment configurations are changed globally as part of an overall mission model in order to allow system architects to change specific configurations according to the intention of the overall mission. Following a block-oriented modeling approach with predefined modules enables architects to keep track of system and environment states at each point of a mission. If, for example, several atomic mission models for an airborne system describe use cases that are applicable during flight only, a preceding system or environment configuration change is used to enable each of the use cases. On the other hand, the same principle can be used to explore inverse what-if scenarios, e.g. in order to explore what happens during simulation if specific conditions for use cases are not provided. Both sets of terms, system configuration and system environment configuration, need to be considered from the perspective of the current level of development, i.e. the current system under design.

Figure 4.32 depicts the relation between system (S_C) and system environment configuration parameter sets (E_C) as part of an EOSS (top box) and configuration change modules as part of a mission model (bottom box). During mission model execution, different configuration parameters of the SuD $s_{C1} \dots s_{Cn} \in S_C$ and its environment $e_{C1} \dots e_{Cm} \in E_C$ are changed by system and system environment change modules according to the intention of the overall mission. Moreover, different levels of system abstraction ($n, n-1, n-2$) are shown.

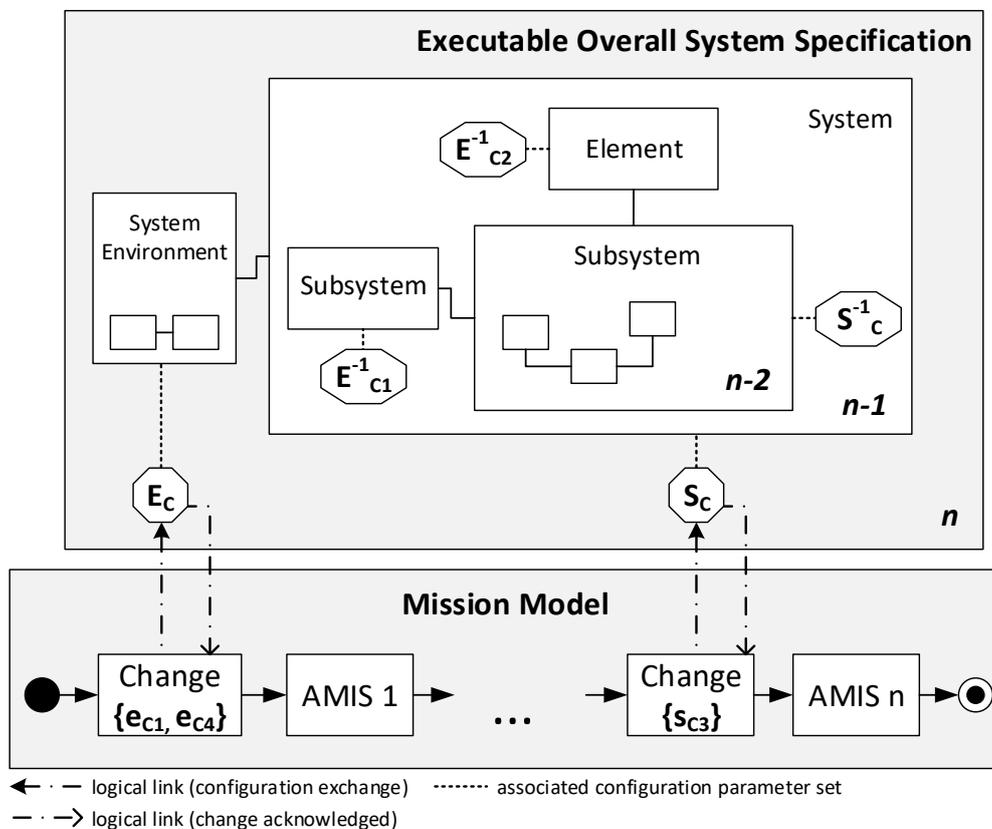


Figure 4.32 – Relation between system (S_C) and system environment configuration (E_C) parameter sets as part of an executable overall system specification with different levels of abstraction ($n, n-1, n-2$) and configuration change modules as part of a mission model

For example, at abstraction level n , a system called *aircraft A1* and several system environment entities, e.g. for ground operations or the physical environment of an aircraft, are considered. System *aircraft A1* has a variety of potential configuration parameters including current speed and height (numeric parameters) or more abstract parameters, e.g. current flight phase (enumerated type). System environment configuration parameters include ambient pressure, possible other aircraft en route a given map, e.g. to simulate possible cross traffic (complex data structure), and specific settings for radio link transmission capabilities of possible ground stations. If the point of view is changed during design, e.g. when considering a specific subsystem of system *aircraft A1* in terms of a SuD, e.g. during subsystem development, our reference for system and system environment shifts. Thus, a set of lower level system configuration parameters S_C^{-1} may be considered for the current SuD in relation with different system environment entities. These entities may be other subsystems or architecture elements of system *aircraft A1* and provide subsets of system environment configuration parameters E_{C1}^{-1} and E_{C2}^{-1} . In general, the level of configuration parameter specialization increases with each lower level of system abstraction. It is also possible to inherit configuration parameters to lower design levels if required.

System and system environment configuration parameters are determined beginning with concept development and refined during mission model development. Especially during the process of requirements analysis and essential use case definition, information on possible configuration parameters can be obtained by analyzing system context and actors. Experiences from similar development projects that have been performed in the past provide another vital source of information. Moreover, configuration parameters may change or emerge during other stages of system development. During overall mission design, the system under design as well as its environment can be viewed as objects with inherent states and characteristics that may change during system application and that can be altered in a way to support different use cases. Changes of system or system environment states may have a global effect on the coupled overall SuD with several sub-mission and atomic mission models that are affected at once. As a result, global system and system environment configuration changes shall only be changed at overall mission level.

If, for instance, example *aircraft A1* is considered as SuD, changes of power or flight states are considered global configuration criteria since both have an effect on a wide variety of coupled subsystems at once. In parallel, local system and system environment configuration possibilities may exist at atomic mission level that are not coupled to any other mission or atomic mission model.

For the purpose of altering system and system environment configuration parameters as part of a mission model, several predefined mission model modules were developed for the associated design environment. Figure 4.33 depicts system and system environment configuration modules for mission models (C1, C2, C3) together with a module for EOSS models (D) that can be linked to a configuration data storage (memory). Instances of modules C1 and C2 can directly be used within mission models to alter system and system environment parameters of a related EOSS model. Module C3 is used as a bridge between custom-developed configuration change modules and executable specifications. System and system environment configuration parameters of executable specification models are stored in the form of memory

components. Dedicated configuration memories exist for the overall system model (containing S_C) and each system environment model (containing E_C or subsets of E_C). Two or more instances of module D, which are linked to global configuration memories, are used as part of an EOSS model in order to execute parameter changes initiated by instances of C1, C2 and C3 modules of mission models. Control flow token input and outputs ports are provided for C1 and C2 for seamless mission model integration. Module C3 is used within custom configuration modules and provides an output port for control flow tokens. Module D has no ports, because it is directly linked to system or system environment configuration memories within executable specification models. Moreover, all modules described can be labeled and documented via label and annotation parameters.

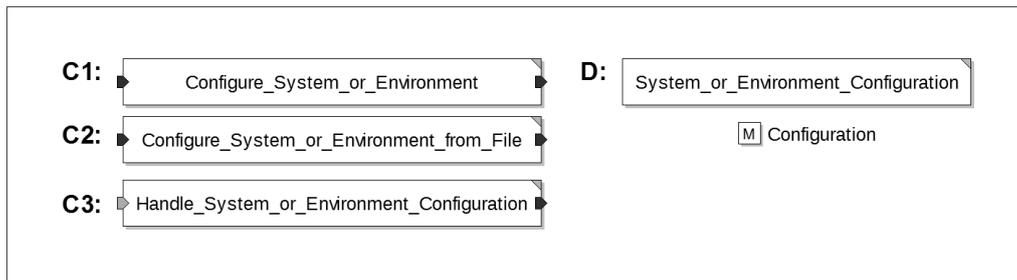


Figure 4.33 – System and system environment configuration modules for mission models (C1, C2, C3) and executable overall specification models (D)

Modules C1, C2 and C3, also referred to as C-type modules, are coupled to respective D-type modules via a global address table at the beginning of overall mission model execution. Information exchange for parameter configuration is one way only from instances of C-type modules to instances of D-type modules. In order to do so, common module parameter *Reference_System_or_Environment_Name* is used by a global address table subroutine which is integrated in all four types of modules. This is similar to mechanism described for atomic mission model actors and quality objective models, including model checking at simulation start-up. Each time one or more configuration parameters of an executable specification model are to be changed, a set of configuration parameters is transferred from a C-type module to the associated D-type module. In this process, the overall control flow is paused and a temporary global address is being created for the currently active C-type module. If all configuration information has been successfully processed by the respective D-type module, this temporary address is used for the transmission of a confirmation signal to the active C-type module in order to resume the overall control flow. This is done in order to guarantee, that necessary preconditions for components of a mission are provided during execution by the associated executable specification model.

D-type modules are integrated part of executable specification models. They are placed at the same level of abstraction used for a specific set of system or system environment configuration parameters that are stored inside a memory model block. Each memory block is typed and initialized with S_C , E_C or subsets of E_C . Two parameters are provided for module D, *Configuration* and *Reference_System_or_Environment_Name*. *Configuration* stores a pointer to a configuration memory inside an executable specification model and is linked via parameter context menu. *Reference_System_or_Environment_Name* determines the name of the system or environment model component that is associated with the respective configuration memory.

Module C1 (*Configure_System_or_Environment*) shown in Figure 4.33 is used as basic component to alter one or more system or system environment configuration parameters when triggered during mission execution. It provides two module parameters, *Reference_System_or_Environment_Name* and *Parameter_to_Change*. Parameter *Reference_System_or_Environment_Name* determines the name of a system or environment component of the executable overall system specification. *Parameter_to_Change* is linked to system or system environment configuration parameter sets S_C and E_C respectively or subsets of E_C . Both are part of the overall data model tree of the respective mission model. Specific configuration parameter values can be accessed and altered by using the respective parameter editor. An arbitrary number of instances of C1 modules can be used within a mission model.

A second way to change system or system environment configuration parameters is provided by module C2 (*Configure_System_or_Environment_from_File*). This module can be used to alter one or more system or system environment configuration parameters during mission execution by reading configuration information from a character-separated values (CSV) file. Three module parameters are provided: *Reference_System_or_Environment_Name*, *Configuration_File_Name* and *Configuration_DS_Type*. As before, *Reference_System_or_Environment_Name* determines the name of a system or environment component of the EOSS. *Configuration_File_Name* is used to determine an input file for configuration changes whereas *Configuration_DS_Type* is used to determine the type of configuration data that is to be read from file. It is thus linked to system or system environment configuration parameter sets S_C and E_C respectively or subsets of E_C . When a C2 module is triggered by receiving a control flow token, it reads an entire line of the specified CSV input file, stores a pointer to the next line and transforms it into a data structure of the type specified by parameter *Configuration_DS_Type*. This data structure is transferred to an associated D module to modify one or more system or system environment parameters within the associated EOSS model. Each time a C2 module is triggered, it will read the next line until the end of the file has been reached. If a C2 module is triggered and has no information to read, i.e. the end of a file has been reached, an error message will be displayed.

In order to create a configuration file for C2 modules, any text editor or any spreadsheet software can be used. In both cases, a plain text file is created that separates entries of each line with tab stops or white spaces. Table 4.1 shows the general structure of system environment configuration CSV files in tabular format. The first line of each configuration file must begin with a #-symbol. This marks the beginning of header information for the data structure that is to be created. Each other entry of the first line denotes a specific member name of the respective data structure (DS) of an associated system or system environment configuration parameter set S_C , E_C or subsets of E_C . In the following lines, specific values are provided for each member.

	1	2	...	n+1
1	#	<i>DS Member Name 1</i>	...	<i>DS Member Name n</i>
2		configuration parameter value 1	...	configuration parameter value n
...				
m		configuration parameter value 1	...	configuration parameter value n

Table 4.1 – Tabular format for system and system environment configuration files

C2 support the generation of data particles with basic data type, composite type and enumerated type. Numbers, strings and enumerations are written in plain text. Values for composite data structures are encapsulated hierarchically within braces with members separated by commas: $\{member_1, member_2, \dots, member_n\}$. Vectors, i.e. array data structures, are encapsulated in box brackets where all entries are separated by colons. However, the first entry is used to determine the length of the respective vector: $[length:entry_1: \dots :entry_n]$. Configuration files in conjunction with C2 modules and control flow nodes enable system architects to automatically iterate large numbers of system or system environment configuration activities during mission execution. Both types of modules C1 and C2 offer a high degree of reusability.

As described in chapter 4.3.1, it is also possible to use custom created model components of type composite model, primitive or finite state machine (FSM) for system and system environment configuration. In this case, instances of module type C3 are used in order to relay customization information to associated executable specification models and to ensure the integration of custom configuration modules within mission models. Module C3 has an input port for receiving system and system environment configuration data structures for S_C , E_C or subsets of E_C . The output port of module C3 produces a control flow token after successful change of system or system environment configuration parameters within an executable specification model. One configurable parameter, *Reference_System_or_Environment_Name*, is provided, which is used to determine the name of a system or environment component of the EOSS. Custom modules for system and system environment configuration parameter changes can be used to determine parallel and complex changes in system and system environment states. As is the case with configuration files, this allows to iterate large numbers of system or system environment configuration activities during mission execution.

When developing mission models, potential side effects during overall system simulation may occur in the course of system and system environment configuration parameter changes. This is the case, when changes in system or system environment states are performed in parallel by different mission model entities, including sub-mission models, atomic mission models and system and system environment configuration models. The impact of configuration changes by local AMIS actors has been discussed at the beginning of this chapter. In addition, overlaps in configuration changes can be introduced by sub-mission models. For instance, if a sub-mission model that includes system and system environment configuration models is executed in parallel with atomic mission models that are affected by these changes. It is thus advisable to determine system and system environment configurations only at overall mission level and to minimize the number of configuration changes within sub-mission models and atomic mission models.

4.3.5 Overall Service Model Development

Based on the overall mission model and top-level requirements from concept design for a given system under design (SuD), an overall service model is developed. An overall service model represents a systematic refinement of the overall set of atomic mission models and consists of two top-down oriented development steps. Firstly, the development of individual service models for each atomic mission model and secondly the development of detailed scenario models for each service model.

An overall service model can be described by a 2-tuple (*Services*, *Scenarios*) where:

- *Services* = $\{se_1, se_2, \dots, se_n\}$ is a finite non-empty set of service models for each atomic mission model: $\forall a_i \in Atomic_Mission_Models \exists !se_i \in Services$ with $i \in N$ and $|Atomic_Mission_Models| \stackrel{!}{=} |Services|$
- *Scenarios* = $\{sc_1, sc_2, \dots, sc_m\}$ is a finite non-empty set of scenario models for each service model: $\forall se_i \in Services \exists service_se_i_scenario_k \in Scenarios$ with $i, k \in N$ and $|Services| \leq |Scenarios|$

In other words, the overall service model for a SuD includes the individual service models for each atomic mission model and each service model describes a structured set of a finite number of associated scenario models. The development process for an overall service model is depicted in Figure 4.34.

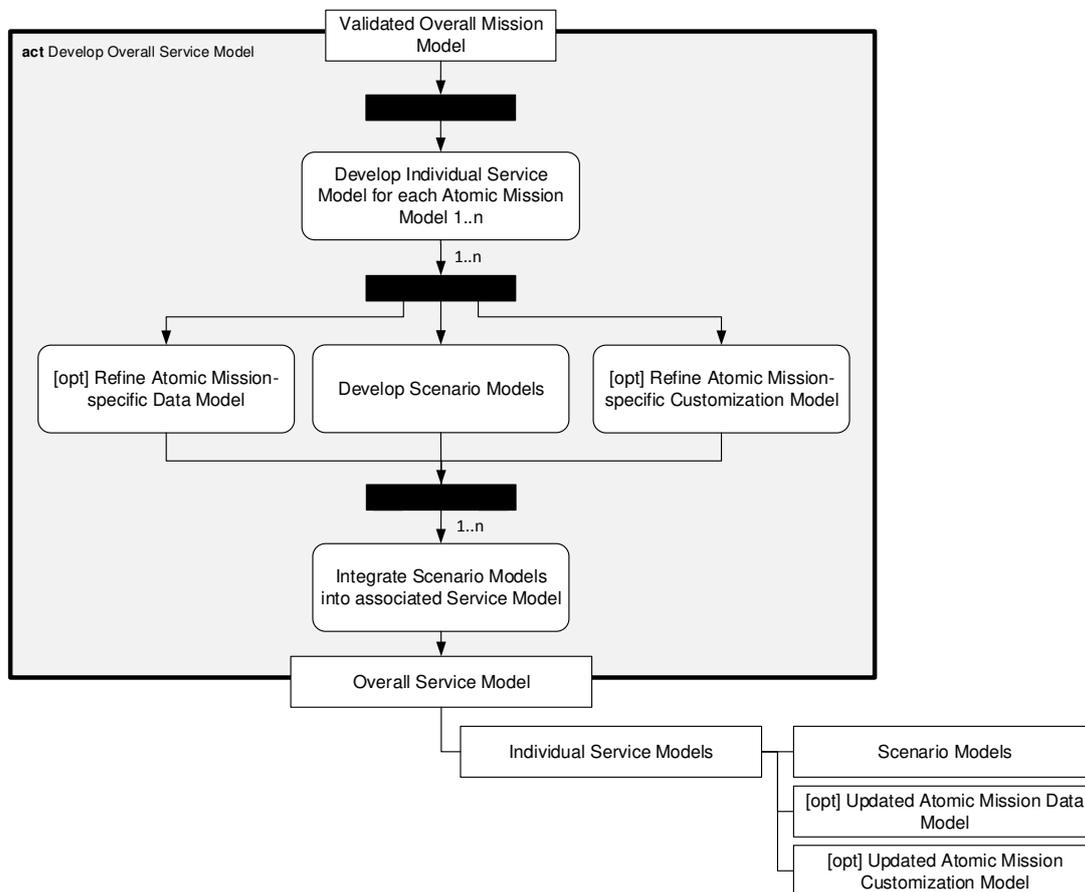


Figure 4.34 – Process for overall service model development

As part of service model development, a structured flow of scenario models is developed. Subsequently, all scenario models of each specific service model are developed. In the course of scenario model development, changes in the data or customization model of the associated atomic mission model may become necessary. At the end of the overall process depicted in Figure 4.34, all individual service models with associated scenario models have been developed and data as well as customization models of associated atomic mission models have been updated.

4.3.5.1 Service Model Development

As described earlier in chapter 4.3, an individual service model is the central node of each atomic mission (AMIS) model and represents characteristics of a use case in relation to different actors. Each use case is characterized by a set of essential, alternative or optional service procedures, called scenarios.

Definition 17 *The **service model** of an atomic mission model is comprised of service scenarios for a use case with specific arrangement. The way in which a service model is structured reflects the typical procedures associated with a use case as intended by relevant stakeholders.*

Service models are specified in the form of directed graphs of scenario models, static quality objective models and control flow nodes. A typical service model has one main scenario and an arbitrary number of side scenarios. In other cases, a service model may provide a set of alternative scenarios that are all of equal significance. For instance, if a service model is composed of a set of scenarios with the same functional and non-functional requirements but different customization model configurations. In this case, a service model is used to describe all stakeholder intended, i.e. useful customization options. Moreover, service models can be used to describe a set of scenarios with same objective but different process operations. A service model may also include a set of static quality objective models that have been described in chapter 4.3.3. This is done in order to determine local non-functional requirements at atomic mission model level, e.g. to determine an AMIS-specific installation space or reliability-related values such as Mean Time Between Failures (MTBF). Again, values from an associated EOSS are acquired by non-functional observer modules.

Service models (SEM) are inherited from executable workflows and can be described by a 12-tuple ($SCENARIOS, QO, C, P_{SEM}, P_C, P_D, I, c_i, EVT, ACT, IN, OUT$) where:

- $SCENARIOS = \{sc_1, sc_2, \dots, sc_i\}$ is a finite non-empty set of scenario models
- $QO = \{qo_1, qo_2, \dots, qo_j\}$ is a finite set of atomic mission model related static quality objective models ($\emptyset \in QO$)
- $C = \{c_1, c_2, \dots, c_x\}$ is a finite non-empty set of control nodes
- P_{SEM} = is a finite non-empty set of configurable parameters
- $P_C \in P_{SEM}$ = is a global parameter containing an atomic mission model-specific customization model
- $P_D \in P_{SEM}$ = is a global parameter containing an atomic mission model-specific data model
- $I \subset C$ = a finite non-empty set of initial control nodes
- $c_i \in C$ is the final node
- EVT = a set of two event elements

- $ACT = \{ade_1, ade_2, \dots, ade_k\}$ is a finite non-empty set of actor data exchange modules
- $IN = \{actor_1in, actor_2in, \dots, actor_nin\}$ is a finite non-empty set of actor data exchange input ports
- $OUT = \{actor_1out, actor_2out, \dots, actor_mout\}$ is a finite non-empty set of actor data exchange output ports

Figure 4.35 depicts the basic structure of a service model. Since service models are connected to actors of atomic mission models, sets of input and output ports are needed for data exchange (solid lines and triangular boxes on left and right).

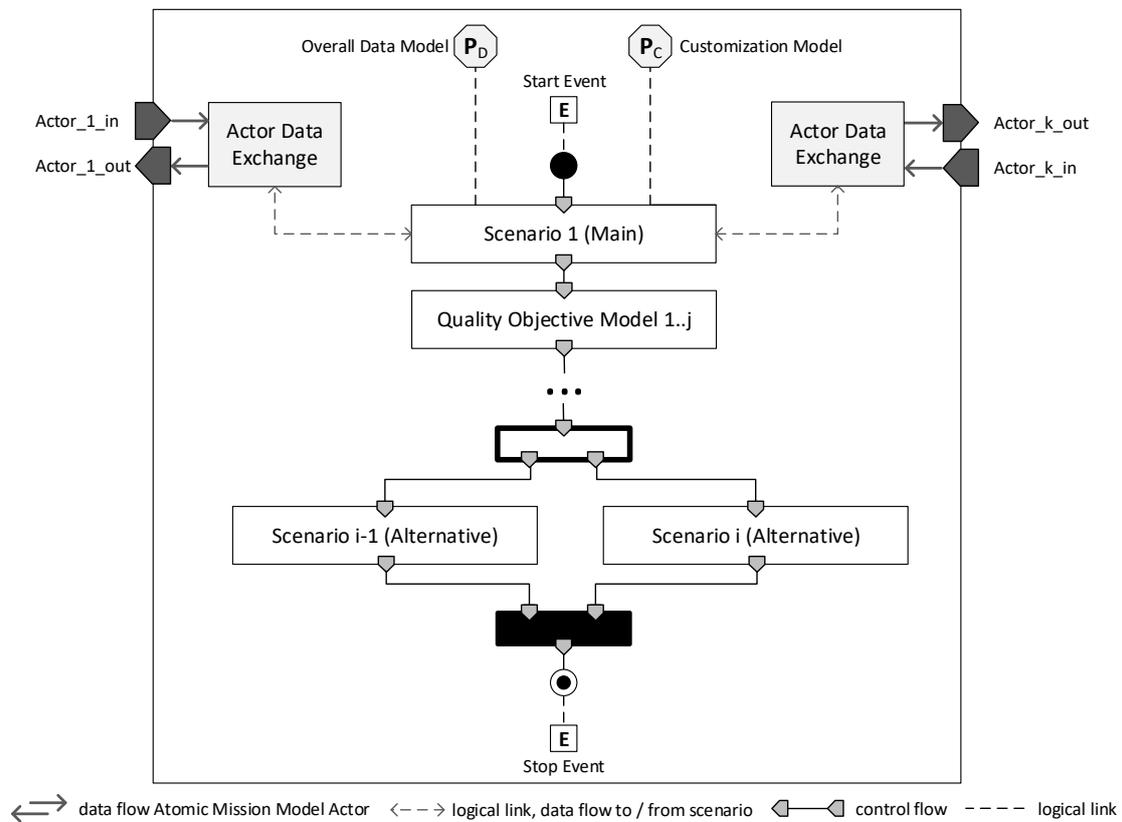


Figure 4.35 – Service model structure

A predefined Actor Data Exchange (ADE) module is used for each actor of the superordinate AMIS model in order to exchange data between actors and service model scenarios. Each ADE module is connected to input and output ports of an associated actor. Moreover, each scenario may establish a logical link (dashed lines with arrows) to one or more ADE modules. Quality objective models have no links to ADE modules and operate according specification (cf. chapter 4.3.3). Customization as well as overall data model of the superordinate AMIS model are directly inherited by a service model via two parameter sets (top). These parameter sets are also linked to scenario models (dashed lines) in order to provide necessary information for scenario development. Eight global parameters are used for documentation purposes and requirements tracing. As with atomic mission models, parameters “*Associated System Name*”, “*Associated System ID*”, “*Associated Mission*

Label”, “*Associated Mission ID*”, “*Associated Atomic Mission Label*” and “*Associated Atomic Mission ID*” are used to link a service model with associated scenario models to a superordinate AMIS model. Parameters “*Service Model Label*” and “*Service Model ID*” are used to identify the respective service model.

Figure 4.36 depicts the developed ADE module, which provides a set of input and output ports for data exchange with AMIS actors. Moreover, it has three configurable parameters, *Associated Atomic Mission Label*, *Associated Service Label* and *Actor Name*. Parameters *Associated Atomic Mission Label* and *Associated Service Label* are directly linked to the respective parameters of the associated service model while *Actor Name* is used to determine the name of the associated AMIS model actor. All three parameters are used by scenario models in order to exchange information.

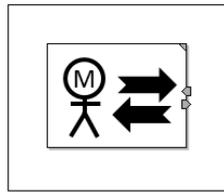


Figure 4.36 – Actor Data Exchange (ADE) module

The control flow of service models is modeled with executable workflows (cf. chapter 4.1). A start event module (top) is linked to all initial control nodes of a service model as well as to the respective event module of the superordinate AMIS model. Likewise, the final control node of a service model is linked to a stop event module (bottom) as well as to the respective event module of the superordinate AMIS model. Event nodes are used to trigger a scenario flow and to indicate scenario flow completion to the superordinate AMIS model respectively. In case a superordinate atomic mission model represents a continuous use case model type, event modules of service as well as AMIS model become ineffective and will not be used. Instead, respective AMIS and service models will start execution with overall mission model execution or when triggered and finish executing with overall mission model termination. Continuous aspects of scenario model execution can be modeled by using appropriate control flow nodes, e.g. timed event control nodes or random timed event control nodes (cf. chapter 4.1.1).

When dealing with atomic mission models that offer a high degree of customization, the possibility of scenario explosion increases for each customization variant that is expressed with a dedicated scenario model. As an alternative to using sets of scenario models for different customization model configurations, customization configuration combinations can also be described within scenario models. In doing so, the risk of scenario explosion can be decreased. The use of customization configuration combinations in scenario models will be described within the next chapter.

4.3.5.2 Scenario Model Development

Scenario models are developed during step two of overall service model development and are derived from conceptual design as well as overall mission model design. As part of this process, atomic mission model-specific interaction and data models can be refined or extended iteratively.

The meaning of scenarios in the context of use case-driven system development has been discussed in chapter 2.3.4.1. Scenario models strongly contribute to the development of functional and architecture models of an overall system model and to the creation of an automated validation process. They interact with an overall system model via system interaction points determined during atomic mission (AMIS) model development (cf. chapter 4.3.2.1), similar to the mechanisms described for test systems with associated *validation points* (cf. [8], [47] and chapter 2.3.5). In the context of this work, the following general definition for scenario models is used:

Definition 18 A *scenario model* determines and validates one specific operational service scenario for an associated service model of an atomic mission model. Each scenario determines functional and non-functional service requirements for a system under design with sequences of concrete events and processes in the sense of system input actions called *stimuli* and expected system reactions called *responses*.

Scenario models inherit data and customization models of an associated atomic mission model and represent a unification of use case-related requirement model and validation model. During overall mission execution, e.g. during automated validation, scenario models use stimuli messages to trigger certain actions or state changes at specific interaction points that are used as input for a system under design. Moreover, a scenario may use stimuli to request current conditions at specific interaction points. A stimulus is defined as follows:

Definition 19 A scenario model *stimulus* is a message in the form of an actor specific data structure with specific values that is sent to one or more actors of the system under design, i.e. an associated executable system specification model, in order to request information, trigger actions or to provoke state changes.

In response to a stimulus or a set of stimuli, a scenario model determines expected reactions or state changes of a system under design that can be observed and evaluated in the form of structured output messages at interaction points, including associated quality and performance properties. Outputs that can be observed at system interaction points are referred to as responses of the system under design. A response may deliver a message with expected content or unexpected content and is defined as follows:

Definition 20 A *response* of a system under design, i.e. an associated executable system specification model, is a message in the form of an actor specific data structure with specific values that is received by a scenario model from one or more actors of an associated executable system specification model in order to evaluate information on functional and non-functional system and system environment properties in response to prior induced stimuli.

The pivotal part of each scenario model is a central scenario flowchart (SFC) module that is modeled with a statechart diagram that is derived from the statecharts that were developed by Harel [156] and finite state machines. It is similar to programming flowcharts or activity diagrams of the Unified Modeling Language (UML) that are

used to substitute use case nodes (cf. reference [365]). Moreover, a scenario model is composed of an arbitrary number of actor signal input (ASI) and actor signal output (ASO) modules that are connected to the central scenario flowchart node, a validation report module and a memory module. Thus, scenario models (SCM) can be described by a 10-tuple $(SFC, ACTOR-SIGNAL_{IN}, ACTOR-SIGNAL_{OUT}, P_{SCM}, P_C, P_D, val, mem, in, out)$ where:

- SFC is a scenario flowchart model
- $ACTOR-SIGNAL_{IN} = \{si_1, si_2, \dots, si_m\}$ is a finite non-empty set of actor signal input modules
- $ACTOR-SIGNAL_{OUT} = \{so_1, so_2, \dots, so_n\}$ is a finite set of actor signal output modules
- $P_{SCM} =$ is a finite non-empty set of configurable parameters
- $P_C \in P_{SCM} =$ is a global parameter containing an atomic mission model-specific customization model
- $P_D \in P_{SCM} =$ is a global parameter containing an atomic mission model-specific data model
- val is a validation report module
- mem is a memory module for validation information exchange
- in is the control flow token input port of the overall scenario model
- out is the control flow token output port of the overall scenario model

Figure 4.37 depicts the general structure of a scenario model. Control flow is modeled top-down, based on tokens. Predefined ASI and ASO modules were developed for the associated design environment that can be connected to the central scenario flowchart module of a scenario model. These modules execute the task of data exchange between SFC and Actor Data Exchange (ADE) modules of the superordinate service model. The data exchange principle based on ADE, ASI and ASO modules has been introduced for reasons of graphical model clarity as well as to include a mechanism for data structure type checking.

The validation information exchange memory depicted in Figure 4.37 is used to store validation information that is produced during scenario model execution. A predefined validation report module is connected to the SFC and validation information memory in order to collect and store validation status information. This information is included within an overall validation report. More information on the process of automated validation is provided in chapter 4.4.2.

Ten global parameters are used for documentation purposes and requirements tracing. As with service models, parameters “Associated System Name”, “Associated System ID”, “Associated Mission Label”, “Associated Mission ID”, “Associated Atomic Mission Label”, “Associated Atomic Mission ID”, “Associated Service Model Label”

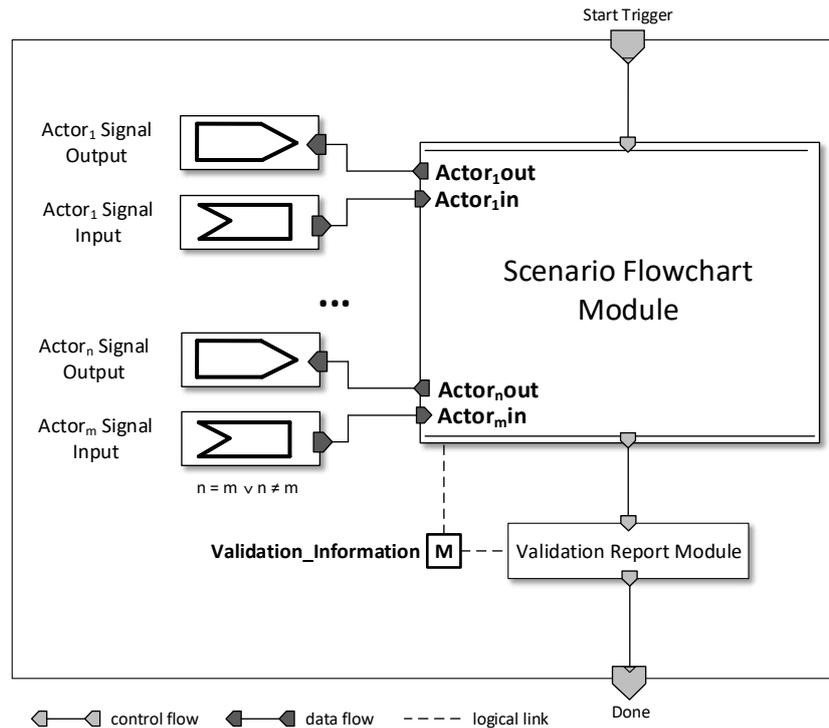


Figure 4.37 – Scenario model structure

and "Associated Service Model ID" are used to link a scenario model to a superordinate service and AMIS model. Parameters "Scenario Model Label" and "Scenario Model ID" are used to identify the respective scenario model.

Figure 4.38 depicts actor signal output, actor signal input and validation report modules. Actor signal output modules have one input port for receiving data structures of type *ACTOR_DS* from an SFC module (cf. Figure 4.25). This type of module is used by SFC modules in order to send data via ADE modules of a superordinate service model to an associated AMIS model actor. Each data structure sent has a data type that is part of the overall data model of the respective AMIS model. Data sent this way eventually provides input data for an actor and system interaction point of an associated executable overall system model in order to trigger a specific action. Following the design of ADE modules described in the previous chapter, ASO modules also provide three configurable parameters, *Associated Atomic Mission Label*, *Associated Service Label* and *Actor Name*. Parameters *Associated Atomic Mission Label* and *Associated Service Label* are directly linked to the respective parameters of the associated scenario model while *Actor Name* is used to determine the name of the associated AMIS model actor. All three parameters are used to exchange information between ADE and ASO modules.

Actor signal input modules have one output port for sending data structures of type *ACTOR_DS* to an SFC module. In this case, an ASI module is used by SFC modules in order to receive data from an associated AMIS model actor via Actor Data Exchange (ADE) modules of a superordinate service model, which relays data from an associated actor of an executable overall system model. As before, each data structure received has a data type that is part of the overall data model of the respective AMIS model. Data received this way provides an SFC module

with information on the behavior and related quality properties of an associated executable overall system model. This information is needed in order to evaluate and validate system reactions in response to SFC induced system input triggers. ASI modules provide the same set of parameters then ASO modules.

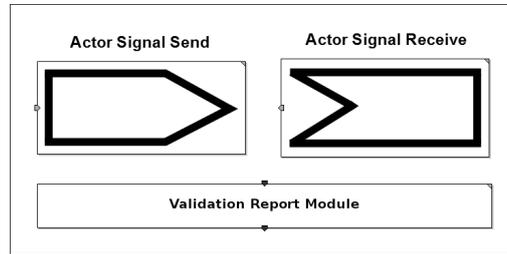


Figure 4.38 – Actor signal output (ASO) module, actor signal input (ASI) module and validation report module

The validation report module for scenario models depicted in Figure 4.38 operates autonomously. It gathers validation-related information during scenario flowchart execution, including information on observed functional and non-functional properties of the SuD during overall system simulation. This information contributes to an overall validation report, generated at the end of overall system simulation. Input and output ports are provided for receiving and producing control flow tokens. Moreover, validation report modules provide the same set of parameters than scenario models and are directly linked to parameters of an associated scenario model.

In the next sections, the development of the central component of each scenario model, i.e. the scenario flowchart module, is elaborated more closely. As described before, SFC represent a form of statecharts. Moreover, SFC are embedded in discrete event domain models. An introduction to statecharts, FSMs and the basics of modeling discrete-event systems with statecharts and finite state machines can be found in literature, e.g. in references [156], [385], [207] and [121]. In general, each SFC consists of a finite number of states, hierarchical states or slave processes, state transitions, special events, internal events, input events, output events, histories for hierarchical states, parameters, memories and a default entrance node. In order to model an SFC, a graphical editor and an action script language is used. State transitions are triggered by events that can be defined in the form of propositional logic formulae. In addition, guard conditions based on propositional logic formulae can be defined for each state transition. Preemptive flags are used to prioritize transitions of hierarchical states. A statechart action language derived from C++ is used to determine actions for transitions, state entry, state exit and for initialization processes executed with the default entrance node.

Each SFC inherits the data and customization model of the superordinate service model and thus, the associated atomic mission model. Figure 4.39 depicts the general steps of a scenario flowchart in relation to atomic mission models and an associated overall system model. Each step depicted in Figure 4.39 may have additional sub-steps within a concrete scenario implementation, depending on the actual design task. The overall control flow is modeled with statechart semantics. Each state can be used to determine an entry or exit action. States are changed via transitions. A synchronous transition is fired quasiparallel as soon as a state with outgoing synchronous transition is reached (ϵ). In addition, a guard condition $[C]$ can be defined

together with an associated action A . All other transitions, called asynchronous transitions, are fired in case a specified event E has occurred. Arbitrary numbers of internal event blocks and memory blocks can be used. An input port T for control flow tokens is used to activate an SFM. Output port D is used to signal completion of the overall SFC. D is triggered by a synchronous transition of the final state, *Scenario Finished*, which resets the overall SFC into its initial state. An SFC can be completed successfully (*OK*) or unsuccessfully (*!OK*).

An external memory called *Validation_Information* can be accessed by all states of the central superordinate state depicted in Figure 4.39 and the last state of an SFC. This is used to exchange validation information with the overall scenario model. In general, SFCs exchange data with system actors at specific system interaction points I via actor-related input and output ports. These ports can be typed with the data model of the associated actor. Any data sent or received is exchanged with actors of an associated executable overall system specification model (EOSS) by means of actors of the superordinate atomic mission model.

Thus, an atomic mission model acts as middle layer for data exchange and data consistency checking. However, it is imperative for system designers to use the specific data model for each targeted actor. Actors are addressed directly by using the actor specific data structure *ACTOR_DS* described in chapter 4.3.2.1. Input data for an EOSS can be created and sent during state entry, exit or during transition firing. Output data received from an EOSS will trigger an input event that can be used to fire transitions. Data received is evaluated and used by states and transitions.

As shown in Figure 4.39, an SFC can be divided into three mayor steps (grey ovals):

1. *Inactive*: In the beginning, an SFC is inactive or idle and waits for an input trigger T .
2. *Active*: When triggered and in state *Idle*, an SFC becomes active. In this superordinate state, a set of five or six sub-steps is performed (white ovals). Each sub-step may provide validation relevant information to the external validation memory:
 - (a) *Initialization*: After becoming active, a set of initialization procedures may be performed, e.g. to initialize memories, data structures or parameter configurations. As part of this process, information may be exchanged with actors of an associated EOSS model, e.g. in order to access EOSS actor characteristics or to determine a specific customization parameter setting for AMIS and EOSS models.
 - (b) *Preconditions*: After initialization, a set of preconditions can be determined and evaluated. Preconditions describe specific states for internal or external actors or system environment configurations of a SuD that are prerequisites for performing a specific scenario, e.g. the authentication of an actor or specific environment conditions. Preconditions are determined and evaluated by using sequences of states and transitions in order to request and evaluate information from condition-related actors of an associated EOSS. In this step, AMIS actor function *GET_CURRENT_STATE* is used in order to request and retrieve information on current EOSS actor states (dashed line with arrows) during overall system execution.

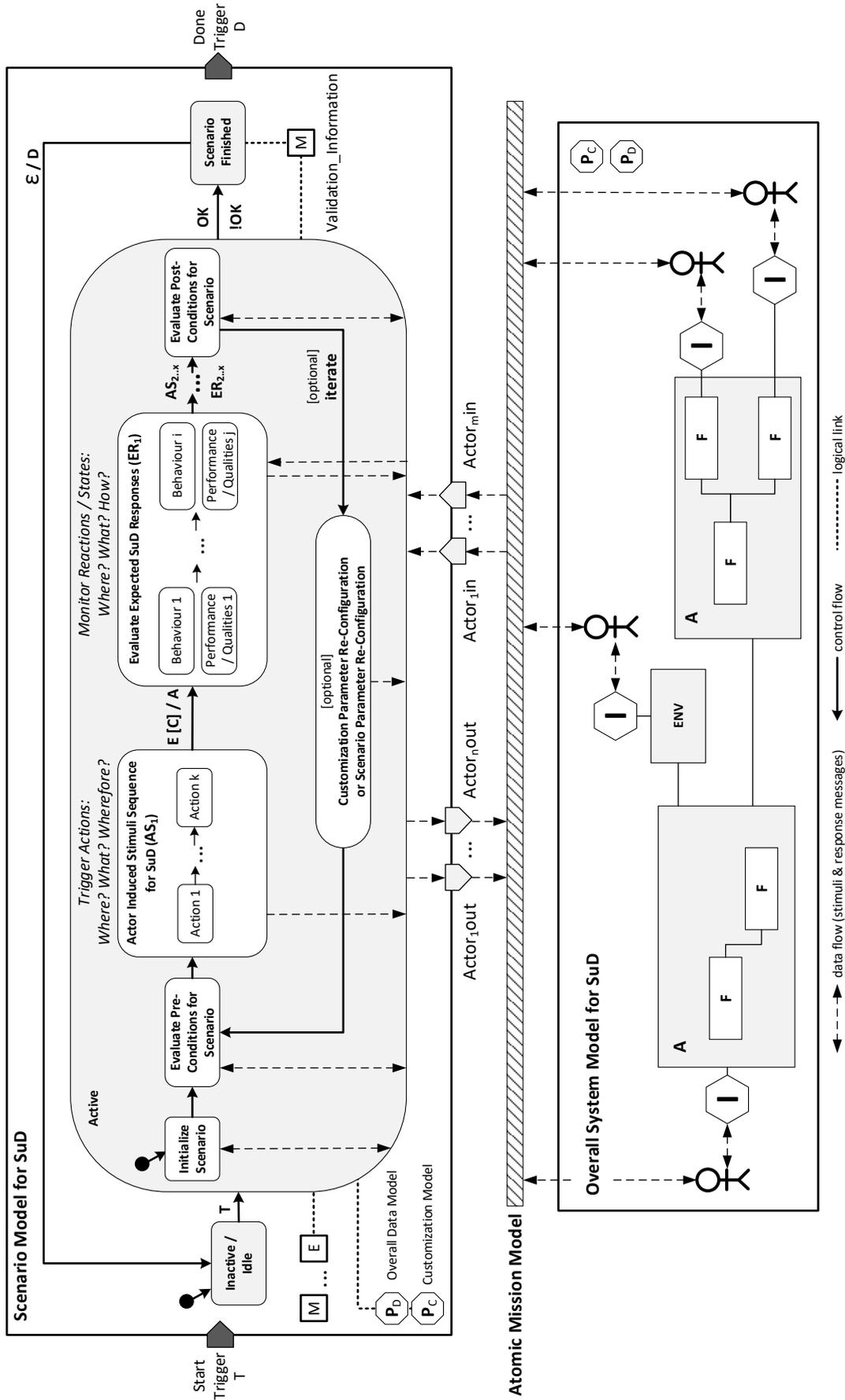


Figure 4.39 – Scenario flowchart (SFC) structure in relation to atomic mission models and overall system model during simulation

- (c) *Actor Stimuli Sequence* (AS_x): An SFC may have an arbitrary number of x actor stimuli sequences. Each stimuli sequence consists of a set of *Actions* $A_1...A_k$. Each action determines a set of sequential and/or parallel stimuli for associated actors of a SuD. Stimuli are generated by determining specific values for instances of the data model of an associated actor and sending them to the respective data output port of the SFC. A stimulus is determined based on answering a sequence of three questions: *Where* at the SuD shall the stimulus be applied? (determine one or more target EOSS actors) *What* is the nature of the stimulus? (determine specific values for actor-related data structures) *Wherefore* do we need the stimulus? (Preparation for the next step).
- (d) *Expected SuD Responses* (ER_x): An SFC may have a number of x ER activities. For each actor stimuli sequence, there is an associated and subsequent ER activity. An ER activity consists of a set of *Behavior Inputs* $B_1...B_i$ and *Performance or Quality Inputs* $Q_1...Q_j$. Each input describes a set of expected SuD responses in reply to stimuli triggered by a preceding actor stimuli sequence. A SuD response is defined to be an input event of a data input port of the SFC that is associated with a specific actor. Moreover, each response includes a data structure instance of the data model of an associated actor with specific values. Response are created by an associated EOSS during overall system execution and are passed on from EOSS actors at specific interaction points. As with quality objective models, responses that reflect non-functional information are determined by using objectives, i.e. value ranges (cf. chapter 4.3.3). During this step, it is also possible to trigger actor outputs in terms of AMIS actor function *GET_CURRENT_STATE* with associated actor input responses, e.g. in order to observe current states of passive actors. Expected SuD responses are determined based on answering a sequence of three questions: *Where* at the SuD shall the response be observed? (determine one or more EOSS response actors) *What* is the functional nature of the response? (determine specific values for an actor-related functional data structure) *What* is the performance or quality of the associated expected functional response? (determine specific values for an actor-related non-functional data structure). It is also possible to determine and measure time-related system qualities for a SuD by using an internal timer event of the overall SFC. For instance, a timer can be used to measure the time it takes a SuD to produce a response to a certain stimulus, e.g. in order to determine a maximum delay objective for audio playback of a public address system for aircraft. As part of this sub-step, timeout events can be determined in order to prevent deadlocks, e.g. in case no SuD response can be observed.
- (e) *Post – Conditions*: After all actor stimuli sequences with associated expected SuD responses have been performed, a set of post-conditions can be determined and evaluated. Post-conditions describe specific states for internal or external actors or system environment configurations of a SuD that shall be present after completing a specific scenario. As with preconditions, this step is performed by using sequences of states and transitions in order to determine, request and evaluate information from actors of an

associated EOSS. Again, AMIS actor function *GET_CURRENT_STATE* can be used in order to retrieve information on current EOSS actor states during overall system execution.

(optional) *Parameter Re-Configuration*: This optional step can be used in order to change the configuration of customization parameters or scenario parameters. Firstly, this step can be used to iterate over a series of scenario executions with different customization parameter values. In this case, a shared memory containing the current customization parameter configurations is used by all previous sub-steps in order to adjust preconditions, actions of actor stimuli sequences, behavior inputs and performance or quality inputs of expected SuD responses as well as post-conditions. It is also possible to have different parallel lanes of AS and ER activity sets that are used depending on current parameter configurations (case selection principle). Secondly, this step can be used to alternate aspects of the overall scenario, e.g. by using different stimuli target and response actors for the same scenario. This is done similar to the exploration of customization alternatives. In doing so, a large number of design alternatives can be described and evaluated by one scenario model (design space exploration).

3. *Finished*: When all activities of step two have been finished, either successfully (*OK*) or unsuccessfully (*!OK*), or an overall timeout event has been triggered (*!OK*), a scenario is ended. As part of this last step, additional validation information is written into the external validation memory. This includes information on the overall status of the scenario after execution, i.e. if all functional and non-functional requirements have been met by the SuD or not. A control flow token is created in order to activate a scenario done trigger *D* while the overall SFC is reset into its initial inactive state.

Although a time-out mechanism is provided for SFCs in order to avoid a deadlock during overall system execution, system designers need to ensure that an SFC is consistent with regard to statechart or FSM semantics.

Since statecharts and FSMs are described formally, model checking can be performed for each SFC. Modeling of actor stimuli sequences and expected SuD responses for a specific use case require a higher level of detail than during the development of AMIS models. As a result, data and customization model of associated AMIS models may need to be updated accordingly. For instance, if additional quality aspects are expected in response to a specific stimuli sequence. Moreover, the extension of data models may also provide new customization possibilities that need to be included within the overall customization model of an associated AMIS model.

Scenarios are intended to describe typical operational sequences with specified system behavior and performances. During scenario execution, especially when executing a set of parallel scenarios due to service or mission model specifications, side effects may occur as part of the execution of an associated EOSS model. These side effects may be wanted or unwanted and can originate in emergent behavior, coupled execution of different scenarios or flawed system models. Scenario models are intended to describe a finite set of wanted and unwanted functional and non-functional properties as part of a use case for a system during system application.

Thus, system responses not related to the execution of a specific scenario are not considered during overall validation, since not all possible interaction points with associated state changes of a SuD are observed and evaluated in parallel. In practice, observation of all possible system interaction points e.g. via monitored system interfaces matrices during overall system execution would not be appropriate. This is because of an exponentially increasing number of possibilities introduced with every scenario executed in parallel. In such as case, each scenario would need to provide validation information for every possible state or parameter change of the overall SuD during overall execution. Especially for complex systems, with a large number of system entities, use cases and configuration parameters, this is a hard task.

As a result, it is crucial to design and validate mission and service models before developing scenario models in order to minimize conflicts between scenarios during overall system execution. In order to support an automated scenario-based validation process with debugging, it is possible to store functional and non-functional properties of all SuD actors and interaction points in a file or database during overall system execution for a more detailed ex post analysis.

4.3.6 Human Machine Interface Concept Model Development

In parallel to the development of an overall service model, human machine interface (HMI) concept models are developed if required. HMIs represent interfaces between human actors and a system under design (SuD) that are required for system operation. Firstly, system-actor interaction possibilities need to be analyzed, especially with regard to human actors. Secondly, physical nature, operability and range of duty of each HMI needs to be determined. Finally, overall structure, usability and design need to be clarified. The concepts and steps of HMI development are not covered by this work. A good introduction to human-machine interaction design is provided by Cooper et al. [89]. Inputs for HMI concept design are top-level requirements from concept development as well as the services that are to be provided by the SuD, i.e. the overall mission model and service model. The development of HMI model components is strongly influenced by the existing definition of system interaction points and actor data models but may also influence both vice versa. Moreover, HMI concept design is driven by system and system environment configuration possibilities as well as customization aspects.

In the context of this work, HMI concept models are represented by graphical user interfaces (GUI) models that are used to interact with executable specifications during simulation. As such, they are intended to be integrated into the executable overall system specification (EOSS) that is developed after the development of an executable requirements specification (ERS). With this virtual prototype (VP), the overall system model can be executed and assessed interactively by different individuals during overall system simulation. During later design stages, peripheral devices connected to the simulation platform can be used to extend the virtual prototype or to perform hardware-in-the-loop (HIL) testing.

Figure 4.40 shows the development of an overall mission model with derived HMI concept model development and integration of HMI model components within an EOSS. HMI components are linked to different parts of an EOSS, including system interaction points I , non-functional observer modules, overall data model P_D ,

customization model P_C as well as system and system environment configuration model S_C and E_C . GUI models can represent a wide variety of visual and audible input/output-related human user interfaces (green box) including switches, indication lights, speakers, displays or multifunctional touch displays to name but a few. GUI model components are used to substitute one or more EOSS actors at specific system interaction points. Based on the overall data model P_D and the specifics of system interaction points (cf. chapter 4.3.2.1), functional and non-functional aspects of system operation can be explored during interactive system execution. Thus, users are able to dynamically provide stimuli for the overall system model and to evaluate system responses (cf. chapter 4.3.5.2).

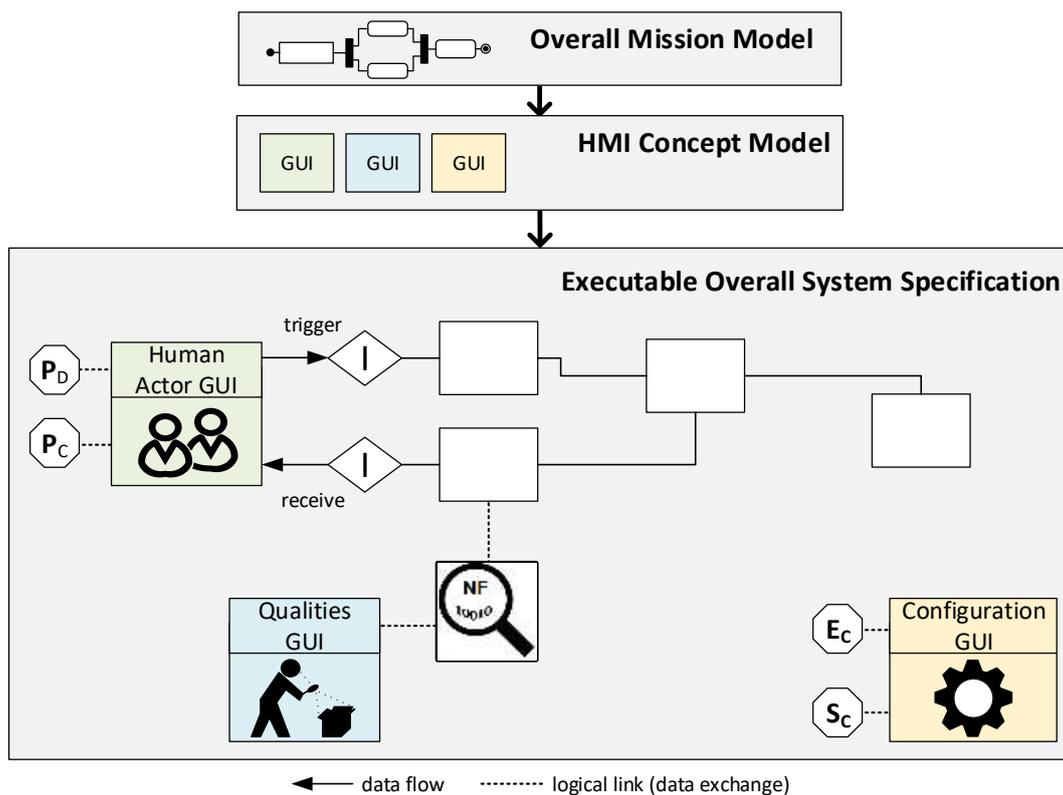


Figure 4.40 – From mission model development to HMI model development and executable overall system specification integration. HMI model components are linked to system interaction points I , non-functional observer modules NF , overall data model P_D , customization model P_C and system or system environment configuration models S_C and E_C

As shown in Figure 4.40, GUI models can also be used to visualize quality properties of the SuD (blue box) or to change system and system environment configurations (yellow box) dynamically during overall system simulation. Quality property displays can directly be linked to already existing non-functional observer modules that were developed during mission model development (cf. chapter 4.3.3). To be able to change system and system environment parameters dynamically during simulation, respective GUI models are linked to overall system and system environment configuration model parameter sets S_C and E_C (cf. chapter 4.3.4). It is also possible to mix, e.g. operational GUIs (green box), with quality property indication GUIs (blue box). GUI models may also be fitted to dynamically adapt structure and function in response to changes of the overall customization model P_C (cf. chapter 4.3.2.2).

Central questions of this design stage include: “What kind of HMIs / GUIs are needed for human actors of the SuD in order to use the system as intended?” and “What fundamental characteristics should each HMI provide (audible, visual, haptic)?” Moreover, inputs from conceptual design are used (cf. chapter 4.2.2). Although intended to be used solely to handle input / output relations between an executable system specification and users, GUI model development may also introduce new functional or non-functional requirements for the SuD. In this case, necessary information has to be fed back into mission model development.

In the context of this work, HMI models are implemented by using the *tool command language* (Tcl) with associated *toolkit* (Tk). Tcl is an open source scripting language that has been developed by John Ousterhout. In combination with the open source *toolkit Tk* that provides a variety of graphical interface elements, Tcl/Tk can be used to create interactive HMIs or GUIs [252]. Moreover, different other toolkits exist that can be integrated in the HMI development process, e.g. the *Snack Sound Toolkit* that was developed by Kåre Sjölander to create audio applications [284]. Reference [252] provides a good introduction to the creation of executable HMIs and GUIs with Tcl/Tk.

The development of HMI and GUI concept models in the context of executable specifications and early design validation has successfully been demonstrated in references [96] and [373]. In the next subchapters, information on the design of GUI models is provided as part of HMI concept model development for ERS and EOSS.

4.3.6.1 Graphical User Interface Model Development

In the case of the development of graphical user interface (GUI) models that are intended to be used in terms of human machine interfaces (HMIs) for an executable overall system specification (EOSS), system architects and system designers need to develop an HMI concept with respect to the intended architecture of the system under design (SuD). This is important in order to determine whether an HMI will be operated by single individuals or multiple users as well as how it will be operated. Moreover, atomic mission models, services and scenarios determined during overall mission model development need to be considered, e.g. in order to determine differences between users of a common multifunctional display. If, for instance, a specific scenario precondition requires user authentication before initiation of any actions, the respective HMI needs to provide means in order to do so, e.g. by integrating a password prompt. Decisions made during HMI concept development may also affect the development of the overall mission model as well as the actual system architecture design that is determined during the development of an EOSS.

In case HMI standards exist for the respective SuD, e.g. ARINC standard 837 - design guidelines for aircraft cabin HMIs [86], these need to be taken into consideration as well. Valuable inputs for this stage of development are also provided from experiences of past development projects.

Graphical aspects of GUI concept models, e.g. custom images for service displays, can be developed with any graphic program, e.g. the open source software *Gimp*. Basic Tcl/Tk elements such as buttons as well as custom designed graphics are arranged and functionally equipped by Tcl/Tk source code. To ease transition to executable Tcl/Tk models, it is also possible to use a wide variety of Tcl/Tk GUI

builders. GUI builders usually offer a drag-and-drop mechanism with context menus for functional configuration. Tools that were used in the context of this work include *Visual Tcl* [11], *SpecTcl* and *GUI Builder* [113]. After a conceptual design has been determined for a specific GUI, resulting Tcl/Tk source code files need to be adjusted in order to include parameters that are used by the associated design environment.

Figure 4.41 depicts the development of a Tcl/Tk module for developing executable graphical user interface models. In general, a basic module called *TclScript* exists that can be configured to have an arbitrary number of input and output ports. Each basic module provides a predefined set of two parameters and integrated source code for executing Tcl files. Parameter *TclFile* is used to determine a default Tcl/Tk file that is used during overall system execution while parameter *TclGfx* is used to determine a default directory for custom designed graphics to be used by the respective Tcl script. Additional parameters can be added if required.

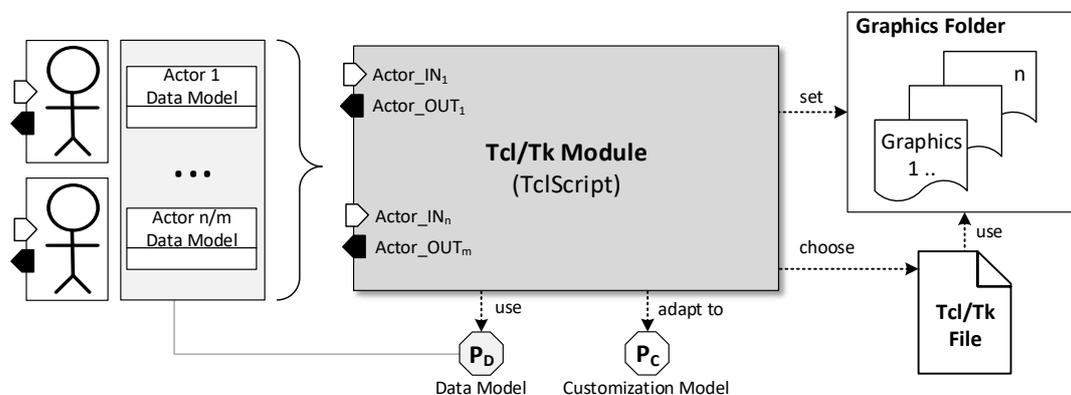


Figure 4.41 – Use of Tcl/Tk for developing graphical user interface models to substitute actor modules of executable system specifications

As shown in Figure 4.41, the number of ports depends on the number of EOSS actors with associated input and output ports. For each actor type that is to be substituted by a GUI model component, the associated data model is used by the respective Tcl/Tk module. Moreover, customization models of related atomic mission models are linked to the respective Tcl/Tk module. With this, the GUI module can adapt behavior and look according to requirements from atomic mission, service and scenario model specification. For instance, by using alternative graphic libraries or Tcl/Tk source code files.

When activating GUI elements during simulation, e.g. buttons, the associated Tcl/Tk module creates data structures of the related data model with specific values. These data structures are sent to the respective EOSS model at the associated system interaction point in the same way stimuli messages are sent by scenario models. Response messages from the associated EOSS model are evaluated by the respective Tcl/Tk module and Tcl file in order to apply necessary changes in the GUI widget during simulation. Information on stimuli and expected response messages can be derived from scenario models (cf. chapter 4.3.5.2). GUI model components should also be tested at this stage of development, e.g. by embedding Tcl/Tk modules in an executable discrete event model that uses probes to observe module outputs.

By introducing visual user interface models that provide means to observe current design properties during overall system simulation, it is possible to create virtual prototypes that enable interactive validation of functional and non-functional properties. More information on virtual prototypes and interactive validation is provided in chapter 4.4.4.

4.3.6.2 Graphical User Interface Customization

Customization plays an important role during user interface design, especially for graphical user interfaces (GUIs) of aircraft cabin systems (cf. chapters 1.1 and 2.1). On one hand, user interface customization comprises aspects of design, often referred to as *look and feel*. For this purpose, different graphical designs can be created, according to customer specification, e.g. by integrating specific logos, layouts and colors. In addition, the behavior of GUI elements such as buttons, displays and menus can be adapted to fit with different customer expectations. By creating different Tcl/Tk modules with associated Tcl script and graphics files, or using additional configuration parameters, different GUI models can be used during simulation via drag-and-drop mechanism.

Already determined customization models from atomic mission (AMIS) model development (cf. chapter 4.3.2.2) can also be used to adapt GUI model appearance and behavior during simulation. This is done by linking Tcl/Tk modules with the overall customization model that is part of each executable overall system specification (EOSS) model.

During human machine interface (HMI) and GUI concept model development, additional functional and non-functional requirements for the system under design (SuD) may emerge that are related to the characteristics of a specific HMI concept. In this case, new requirements are validated with associated stakeholders and are fed back into mission model development if necessary.

4.3.6.3 Non-Functional Parameter Visualization

In addition to functional and non-functional system specification aspects related to system services, global non-functional parameters, i.e. quality properties of an executable overall system specification (EOSS) model, e.g. overall cost or power consumption, can be also be visualized and interactively validated during overall system execution by using predefined or customized graphical user interfaces (GUIs). By default, a modified quality objective model module may be used in order to display and evaluate system quality characteristics (cf. chapter 4.3.3).

Figure 4.42 depicts a predefined quality objective model module that is used for displaying and evaluating overall system properties dynamically during simulation (online graph) and a posteriori after simulation (post-simulation graph). In accordance with quality objective models, three parameters are provided (cf. chapter 4.3.3). Parameter “*System Name*” is used to determine the associated system under design while parameter “*Non-functional Parameter Name*” is used in order to specify a unique name for the quality property that is to be evaluated. To determine an objective for the non-functional system requirement that is to be observed, parameter “*Objective*” is used. All parameter values can directly be derived from quality objective models that were developed during mission model development.

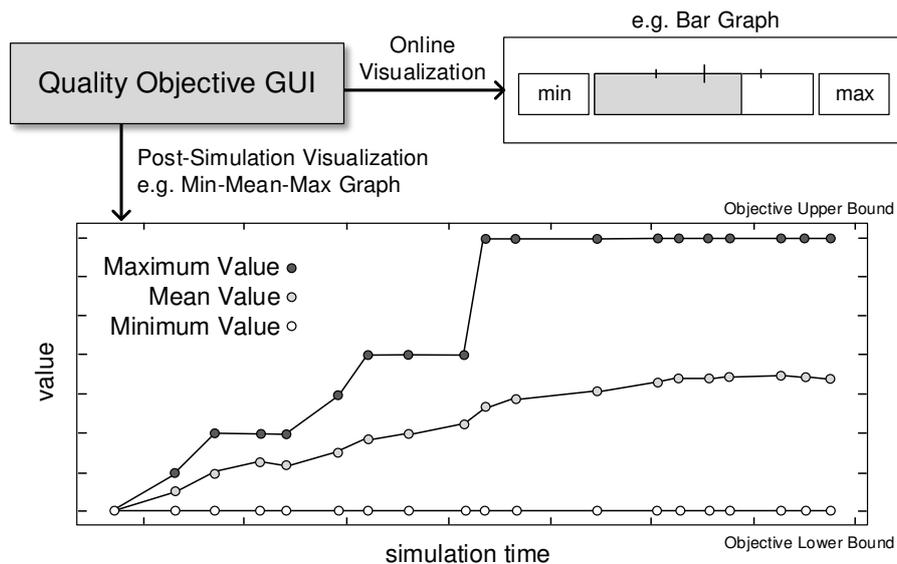


Figure 4.42 – Adapted quality objective visualization module with corresponding displays during simulation (right) and after simulation (bottom)

Quality objective visualization modules can be placed anywhere within an EOSS model and automatically gather needed information by accessing associated non-functional observer modules that are already part of the overall system model. Moreover, the module type depicted in Figure 4.42 provides an output port that can be used to provide information on a specific quality property to custom designed GUI models. In this case, custom GUI model development is similar to the design of user interfaces for system services depicted in Figure 4.41, whereby actor inputs are replaced by quality objective module inputs and data as well as customization model links may not be required.

Online bar graphs that are created by quality objective modules during overall system simulation show current values for a specific quality property, e.g. overall power consumption. Moreover, minimum and maximum boundaries are shown that result from the objective of the respective quality property. With this, users can validate if values for system quality properties are within the required budget or range. For more detailed post simulation analysis, a minimum-mean-maximum graph is plotted at the end of simulation. This graph shows the characteristics of a specific quality objective over time. As shown in Figure 4.42, three different lines are drawn for parameter values observed during simulation in relation to two black lines that indicate upper and lower objective bounds. The lower line of the graph shows the overall minimum of values determined during simulation. The line in the middle shows the development of the mean value over time while the top line is used to show the development of maximum values during simulation. Together, all three lines represent a bounded range for a specific system quality property during overall system simulation. In the example graph shown in Figure 4.42, all values observed during simulation are within the determined objective boundary. Thus, the given objective is regarded to be attainable. In other cases, values outside a given objective range indicate a failed quality objective.

However, when evaluating static quality objectives, e.g. overall system weight, online and post-simulation graphs will also both be static, i.e. they only show fixed

values. In this case, all three lines of the post-simulation graph will overlap since no parameter value variation is observable. Thus, static quality objectives should be visualized, e.g. with basic textual simulation displays.

4.3.6.4 System and Environment Configuration

Graphical user interfaces (GUIs) for system and system environment parameter alteration are, like non-functional parameter visualization GUIs, not part of the overall set of human machine interfaces (HMIs) that are required for operating the system under design (SuD). Instead, these GUIs are important to enable interactive simulation and validation of executable overall system specification (EOSS) models. Since configuration models for system and system environment have already been determined during overall mission model development (cf. chapter 4.3.4), necessary configuration parameters, data structures and library modules are already available.

In terms of configuring system and system environment parameter values for interactive system simulation, two general possibilities exist. Firstly, system and system environment parameters can be directly changed within the configuration model of an EOSS model before overall system execution. In doing so, no additional GUIs are needed. However, only one specific configuration can be evaluated during each simulation run.

Secondly, in order to provide interactive configuration possibilities during overall system simulation, GUIs for system and system environment configuration can be developed beginning with HMI concept model development. By being able to change system and system environment configurations dynamically during overall system execution, different service scenarios with associated system and system environment configurations can be explored on demand during one simulation session. Figure 4.43 depicts how GUI models for interactive system and system environment parameter alteration for EOSS can be developed and used.

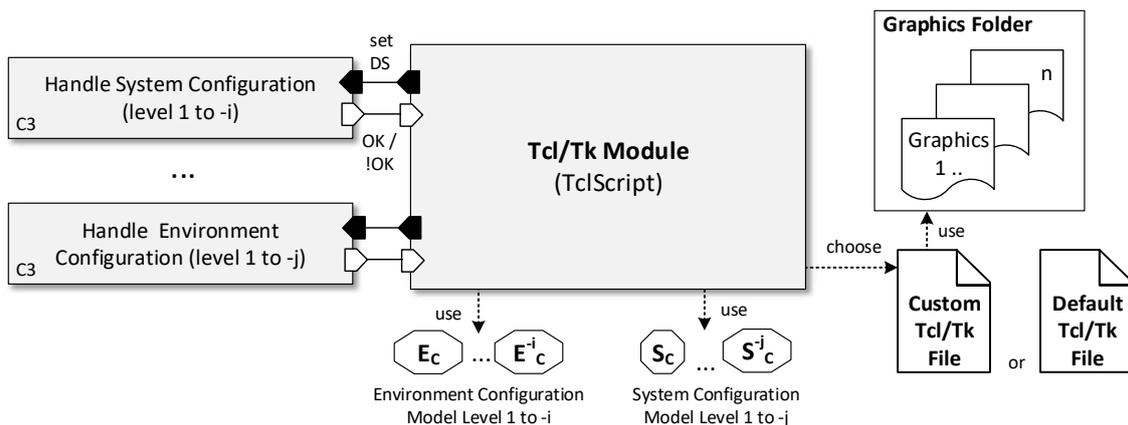


Figure 4.43 – Developing graphical Tcl/Tk-based user interface models for interactive system and system environment parameter alteration of executable system specifications

Tcl/Tk configuration modules provide graphical means to create data structures (DS) with specific values for changing configuration parameters of system and system environment models (S_C and E_C) or submodels at levels 2 to i and j respectively (cf. chapter 4.3.4). These data structures are handed over to system and system

environment configuration C3 modules that perform the task of system and system environment configuration adjustment of the associated EOSS model during overall system simulation (cf. chapter 4.3.4).

In addition, the control token output port of C3 modules can be used to provide feedback on the success of induced configuration changes to the associated Tcl/Tk module during simulation. Custom designed Tcl/Tk modules can be developed similar to the development of interactive user interfaces described in chapter 4.3.6.1. As with quality objective visualization modules, system and system environment configuration modules can be placed anywhere within an EOSS model.

In the case of complex composite configuration parameters, it is possible to create specialized custom GUIs that enable users to specify complex environment properties and objects. Liebezeit for example developed a dedicated mission model parameter editor called *MLEditor* with associated visualization during overall system simulation called *MLVisor* for autonomous underwater vehicles (AUV) [198]. With *MLEditor*, it is possible to determine composite mission, system and environment parameters, e.g. to determine an AUV route based on a map with waypoints and different obstacles.

Since it is possible to determine GUIs for all system and system environment configuration parameters, a large set of GUIs may be used in parallel during overall system simulation. As a result, it will be possible for users to determine and analyze large numbers of different parameter combinations during simulation. Some of these combinations may not be useful or feasible for the SuD, e.g. if two opposed configuration parameters are set. For example, if an aircraft is considered in flight, a ground power plug cannot be set to be operative at the same time. To avoid opposing or non-feasible configurations, two general possibilities exist. Firstly, users need to take care when changing system and system environment configurations during overall system simulation with respect to the overall mission. Secondly, configuration GUIs can be designed in a way to avoid configuration mismatches, e.g. by grouping coupled parameters or by automatically deactivating other parameters when activating a specific configuration. However, configuration GUI restrictions need to be developed with regard to the overall mission model. However, in terms of testing during later design stages, it may be necessary to provide an unlimited set of configuration possibilities.

4.3.7 Requirements Documentation and Tracing

In the course of the development of an executable requirements specification (ERS), it is important to be able to document and trace requirements with respect to top-level requirements from concept development. Requirements from executable overall system specifications (EOSS) need to be documentable and traceable with respect to ERS or concept requirements. As part of the development process for executable specifications, top-level requirements as part of a conceptual mind map are linked to documented mission, service as well as overall system models and vice versa.

Many design tools, e.g. MLDesigner, already support the automatic creation and export of hypertext-based documentation files for each model instance. Each documentation file contains links to related objects, textual descriptions and visualizations, e.g. graphics, for model properties, parameters, ports and other model instances. It

is also possible to set an arbitrary number of annotations within model instances, e.g. to write notes during modeling, that can be included optionally within the overall documentation file [222].

For requirements documentation and tracing purposes of ERS and EOSS models, an auto documentation function is used as part of the associated design environment developed in this work. Each instance of an ERS and EOSS model provides a set of four default documentation parameters called “*Author*”, “*Version*”, “*Description*” and “*Documentation*”. Two other parameters, “*Date*” and “*Time*”, are added to each documentation file automatically. *Author* is used to determine one or more authors for each model instance while parameter “*Version*” is used to keep track of different versions of a model instance. Parameter *Description* represents a text-type parameter field that is used to describe the characteristics of each model instance and for generating tool tip text when using a specific model instance as part of another high-level model instance. Information provided through this parameter is also displayed on top of the created HyperText Markup Language (HTML) documentation file that is stored in the respective user directory. Similar to parameter *Description*, parameter *Documentation* provides documentation capabilities in terms of textual inputs but is extended with the capability to use HTML constructors as part of documentation text. Documentation files are created automatically for every model instance. Moreover, it is possible to create and export documentation files for single instances as well comprehensive files at overall system or mission level. Since documentation files use HTML, it is possible to browse through documentations of different coupled model instances top-down or bottom-up. In doing so, it is possible to trace a specific scenario model beginning at overall mission level and vice versa.

At overall mission level, HTML documentation files include information on the overall control flow, sub-mission, atomic mission, quality objective and system as well as system environment configuration models. Atomic mission model documentation includes information on actors and the underlying use case, i.e. the underlying service model. This includes documentation for data and customization models. Documentation for quality objective models include detailed descriptions on associated non-functional parameters and objectives for the system under design (SuD), including data types and value boundaries. System and system environment configuration model documentation includes information on specific values of configuration parameters to be changed together with the associated system or system environment configuration models. In addition, custom designed system and system environment configuration models may provide more detailed information on each step of configuration adjustment.

Service models, as part of atomic mission models, include information on overall control flow as well as on all scenario model instances used. At scenario model level, detailed information is provided on functional and non-functional requirements for the SuD. This includes scenario model-specific parts of the overall data and customization model. Moreover, the specifics of scenario flowchart models are provided, including descriptions of stimuli and expected responses for the SuD.

As indicated before, instances of ERS and EOSS models can be linked to higher level specifications, i.e. the conceptual design that was created in the form of a mind map (cf. Figure 4.13). Since each node of a conceptual mind map can be linked to any object, links can also be established to documentation instances of any executable

specification. Thereby, executable specification models or model documentations can directly be accessed from within the conceptual design mind map. In doing so, requirements can be traced between conceptual design and executable specifications. It is also possible to update existing conceptual designs by extending already defined human machine interface (HMI) concepts with more specific HMI concept models that are intended to be integrated within EOSS models (cf. chapter 4.3.6). Figure 4.44 depicts how conceptual design mind maps are extended with links to external objects, including already determined concept designs, HMI concept models and HTML documentation of ERS and EOSS model instances. It is also possible to include a link to a mind map or mind map node within ERS or EOSS model components by using the *Documentation* parameter which is common to all models.

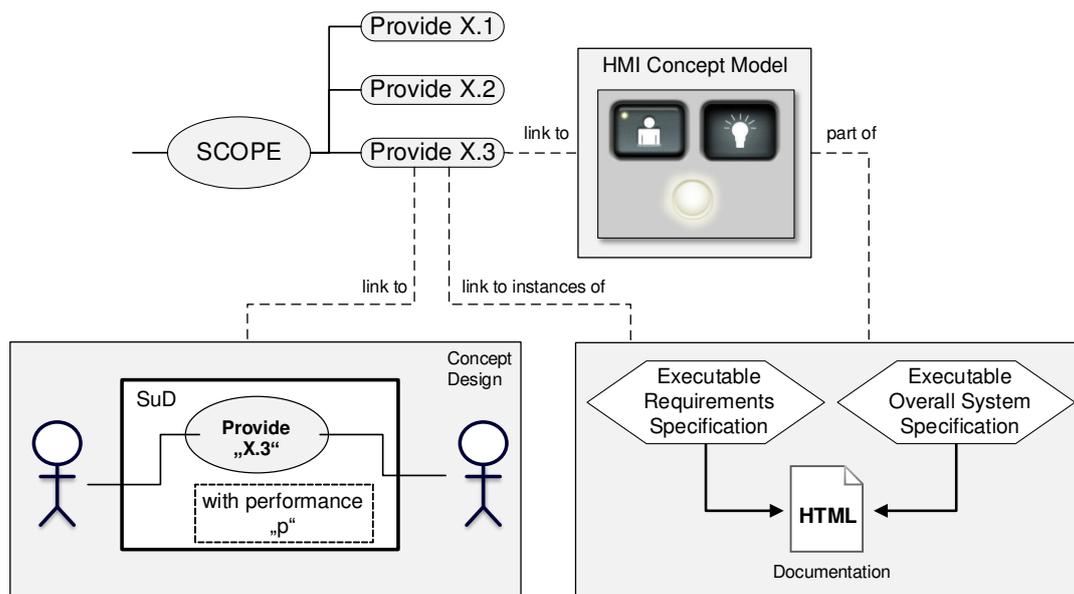


Figure 4.44 – Extended conceptual design mind map with already existing concept designs and additional links to external objects, including HMI concepts and HTML documentations for ERS and EOSS model instances

4.3.8 Executable Requirements Specification Validation

Executable requirements specification (ERS) development is part of the earliest development steps of system design and is closely related to conceptual design. Moreover, this stage of development utilizes experiences from previous development projects and direct stakeholder inputs. Therefore, it is also necessary to validate different aspects of an ERS against the validated conceptual design and, in addition, against stakeholder needs and experiences. In order to do so, stakeholder representatives need to be involved during different validation stages, e.g. customers, system and certification experts or marketing analysts [96], [373] and [130]. Having a validated executable requirements specification with integrated mission and service model is crucial for the successful development and automated validation of an executable overall system specification (EOSS), since mission and service model determine intended overall application and performance of any system under design (SuD). As a result, developing a valid ERS strongly contributes to minimizing overall product uncertainty early during design, thus minimizing the overall risk of

system development (cf. chapter 2.2). In addition, validated HMI models that are developed in accordance with mission and service profiles provide interactive EOSS validation and user training capabilities. Validated ERS can also be re-used for the development and validation of future projects with similar scope.

Figure 4.45 depicts the flow of development and validation from conceptual design to ERS development together with different validation paths. Aspects of an ERS that are validated against stakeholder needs and experiences are marked with blue circles while aspects that are validated against the conceptual design are marked with green circles. In addition, human machine interface (HMI) concept development is shown with respect to the overall development and validation flow.

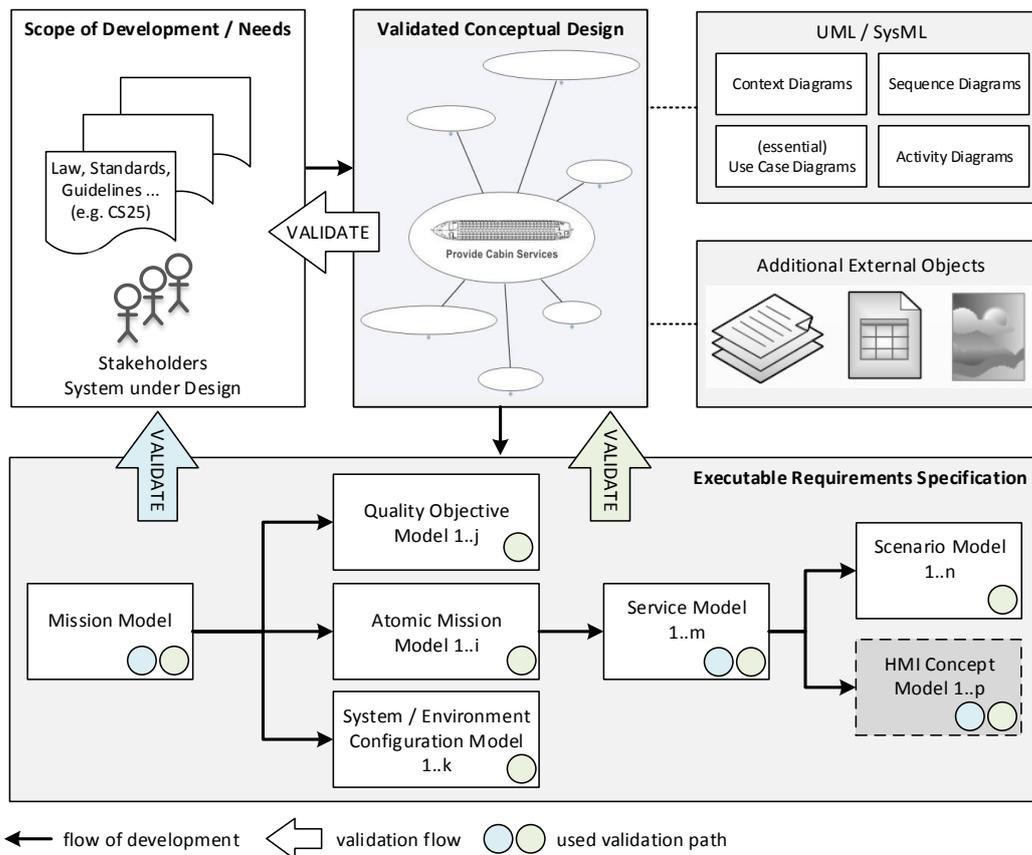


Figure 4.45 – Development process and validation flow for executable requirements specifications against concept design (green) or with stakeholder participation (blue)

Although the development and validation process of ERS has been described sequentially in the previous sub-chapters, the overall process is considered part of an iterative and phase-oriented development process, e.g. the V-Model described in chapter 2.1. Therefore, parts of the process will be iterated on in the course of each development project, e.g.

because new non-functional requirements or system applications arise in terms of operational scenarios, i.e. missions, or use cases, i.e. atomic missions, with associated services.

Mission models unite functional and non-functional aspects with regard to overall system application and performance, which is primarily determined by stakeholders and system context. This is why mission model validation is performed against the

conceptual design together with the aid of stakeholders and based on system experience. System experience is, among other things, based on user knowledge, regulations, constraints, previous or similar developments by one or more system suppliers, market trends as well as forecasts and excitement factors (cf. chapter 4.2.2).

Most important aspect during overall mission model validation is to check that a valid executable workflow has been defined whereby each mission model must be consistent, feasible, executable and testable. Mission model procedures need to correctly reflect typical and intended operational scenarios for the SuD. The same applies to service and scenario models. All mission model nodes and relations between nodes need to be valid, especially with regard to sequential and parallel execution of atomic mission (AMIS), quality objective (QO) as well as system and system environment configuration models. Especially in terms of atomic mission model arrangement as well as system and system environment configuration, this task is to be performed in collaboration with stakeholder representatives. It is also important to verify, that top-down mission development and bottom-up service integration has been performed correctly. Thus, reviews need to ensure that each AMIS model is extended by the associated service model that in turn includes all intended scenario models.

Atomic mission model validation is mainly performed against concept design and needs to ensure that all required use cases and misuse cases from concept design are covered by atomic mission models. During this process, the following question need to be examined for each AMIS model:

- Does the atomic mission model represent the original use case correctly?
- Have all necessary actors and system interaction points been determined?
- Does each actor possess a use case-related data model according to its purpose?
- Does the overall data model contain all necessary information in order to provide the associated use case?
- Does the overall customization model contain all necessary customization parameters?
- Have all data model and customization model parameters have been defined and typed correctly?
- Have all data model and customization model parameters been determined with valid bounded value ranges?

In terms of overall system performance, the validation of non-functional requirements, i.e. quality requirements, is performed against top-level requirements of concept design. Quality requirements from concept design need to be covered by respective quality objective models with associated non-functional observers. The most important questions to be answered during validation include:

- Does the quality model represent non-functional top level properties correctly?

- Have all static and continuous quality objectives been determined correctly?
- Have all non-functional parameters in terms of objectives and constraints been determined with valid bounded value ranges?

Configuration models shape the overall flow of mission models by enabling use cases and adjusting system as well as system environment states. Thus, initial as well as intermediate system and system environment configuration steps are checked for validity with regard to the overall mission. Major questions to be validated include:

- Does each configuration step with associated system or environment states match the current state of the overall mission (mission profile)?
- Does the configuration model provide all necessary conditions that are required for other mission model elements at specific stages of mission execution?
- Have all necessary configuration parameters been determined and have all parameters been typed and set correctly?
- Are all custom configuration models or configuration files well-defined and have they been determined correctly?

Service models are validated with stakeholder representatives as well as against concept design. Stakeholder involvement is required in order to validate that each service model, which is linked to a superordinate atomic mission model, is represented by a valid arrangement of primary, secondary and optional service scenarios and quality objective models. Like mission models, service models need to constitute a valid executable workflow which is consistent, feasible, executable and testable. All service model nodes and relations between nodes need to be valid, especially with regard to sequential and parallel execution of different scenario models. Moreover, a valid service model shall correctly reflect typical and intended sets of use case scenarios for the SuD. It is also important to verify, that necessary actors are interfaced and that the associated data and customization models have been embedded correctly.

Although scenario models, including scenario flowchart modules, are validated mainly against concept design, it is also possible to involve stakeholders at this stage of validation. This may be helpful in order to validate that each scenario reflects the intended process of service provision for each use case, including stimuli and response definitions. Scenarios flowchart modules need to be defined according scenario model specification in order to be consistent, feasible, executable and testable. In order to validate scenarios by means of model execution, it is also possible to develop scenario specific environment test models that simulate necessary aspects of the SuD that are associated with the respective scenario. During scenario execution, it is possible to use a statechart animation mode in order to track the execution of states and transitions to evaluate that a valid scenario has been determined. Among others, the following questions are validated for each scenario model with associated scenario flowchart module:

- Does the scenario model represent the associated atomic mission model correctly?

- Have all necessary actor signals been determined?
- Do stimuli and response definitions match the overall scenario intention?
- Have data and customization models been integrated correctly?
- Does the overall data model contain all necessary information in order to provide the associated scenario?
- Does the overall customization model contain all necessary customization parameters?
- Have all applicable and useful customization model combinations been determined?

For HMI and graphical user interface (GUI) models, validation is performed against early definitions from concept design, usability requirements as well as the intended application of each HMI (use case-related). Moreover, customers and other stakeholders need to be consulted in order to validate overall *look and feel* as well as customer specific customization properties, especially for complex user interfaces like multipurpose touchscreens. If all validation activities have been performed successfully, the development process continues with the development of an executable overall system specification.

4.4 Executable Overall System Specification Development

With regard to the original mission-level design approach by Schorcht [323], validated executable requirements specifications (VERS) are used early during systems design phases in order to determine and validate what mission / application and services the system under design (SuD) shall provide together with non-functional properties, performance parameters and constraints. Subsequently, the VERS is used to develop and validate an executable overall system specification (EOSS), also referred to as executable technical specification, which determines how the SuD will provide specific functions with associated performances. An EOSS is based on an optimized physical system architecture, in order to accomplish all missions and services while meeting non-functional objectives and constraints. Following the meet-in-the-middle approach, validated and verified components, e.g. from previous development projects, can be re-used and are integrated bottom-up during this stage of development. Figure 4.46 depicts the overall development and validation process of EOSS which leads to detailed subsystem and component design at the end of the process.

In the beginning of the overall process depicted in Figure 4.46, an executable overall system specification model is developed that unites function, architecture and environment model (system context). The importance and feasibility of combined functional and system architecture design has been described, e.g. in references [215], [125] or [128]. During this step, already existing and validated model library components from previous developments can be re-used while unavailable model components, that need to be developed and re-integrated during detailed design, are

specified at system level. During the next step, two different validation paths are executed. Firstly, an automated validation of the EOSS is performed based on the integrated mission model of the VERS. Secondly, EOSS and the human machine interface (HMI) concept model from VERS development are combined in order to form a virtual prototype (VP). This virtual prototype is used for interactive validation with different stakeholder representatives. Later during development, VPs can also be used for performing user-related tests and early user training.

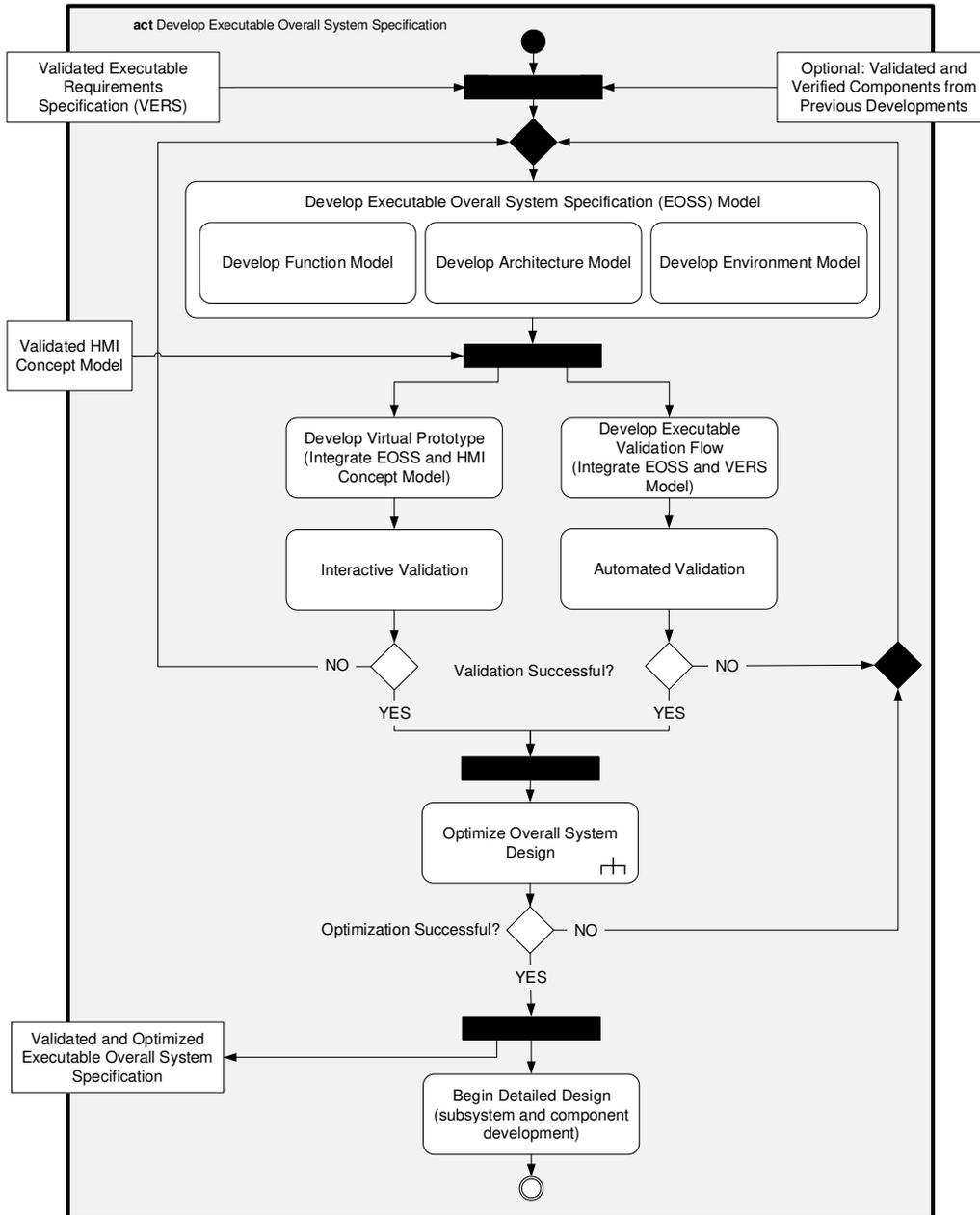


Figure 4.46 – Process for executable overall system specification development

After the development of a validated EOSS, the overall system specification model is optimized and re-validated with regard to non-functional top-level criteria, e.g. in order to minimize overall weight. In case more than one criterion shall be used for system optimization, Pareto optimization is performed. In order to achieve an

optimal design, it may be necessary to adapt and change aspects of the EOSS and to iterate the optimization process until a feasible solution has been found.

At the end of the development process depicted in Figure 4.46, an optimized and validated EOSS has been developed which is used as basis for detailed development. Figure 4.47 depicts the process of detailed development and validation. During this process, subsystems and components that are not already available in the form of executable specification model components are developed in detail.

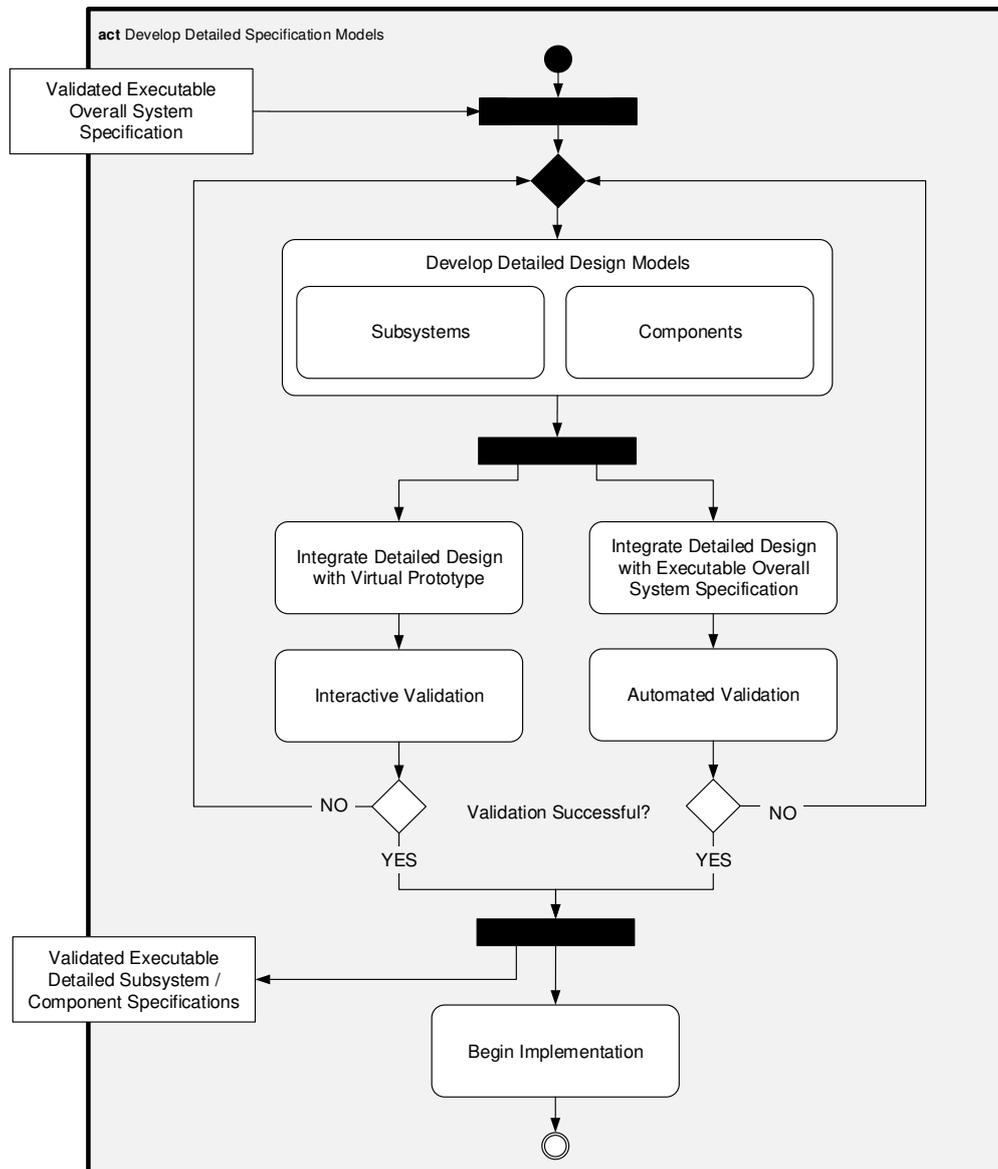


Figure 4.47 – Process for detailed system development and validation

In other cases, already available model components may need to be adapted for the current SuD. At this stage of development, overall development is divided into different subsystems that are developed by either manufacturer internal departments or external system suppliers.

When finishing detailed development, it is necessary to re-integrate detailed model components within the overall EOSS and virtual prototype with subsequent re-

validation. This is sometimes referred to as virtual integration and is required in order to validate detailed designs in the presence of the coupled overall system architecture, i.e. the integrated executable overall system design. As before, two different validation paths can be executed. An automated validation that uses VERS and EOSS model and interactive validation which uses virtual prototypes. After successful validation of the integrated EOSS, the process of system implementation is initiated. Since the development of detailed design models follows the development paradigm for executable specification models, which is described in chapter 4.4.1, this process is not covered in detail by this work.

As shown in Figures 4.46 and 4.47, extensive and repeated validation activities need to be performed during EOSS and detailed development. By using automated validation based on reusable executable workflows, validation efforts required after each system specification development step are minimized. Moreover, overall design uncertainty is minimized since validated VERS models and virtual prototypes are used to validate overall specification design. During later stages of overall system development, during implementation, integration and test, developed subsystems and system components are verified and tested against (detailed) executable specification models. This is done in order to verify that hardware and software have been developed and implemented according specification (“*Are we building the system right?*”). For more information on model-based verification, please refer to reference [254].

4.4.1 System Specification Model Development

In this chapter, the development of executable system specifications (ESS) is elaborated more closely. In the case of an ESS that represents the highest level of abstraction for a system under design (SuD), i.e. the overall system level, this model is referred to as executable overall system specifications (EOSS). Since the underlying creation of executable computer models, e.g. based on block-diagrams with domain-specific models, is state of the art and well documented in literature, the fundamental aspects of such model creation are not covered in detail as part of this work. For more information on the creation and simulation of domain-specific computer models please refer to, for instance, references [27] or [121]. Different terms are being defined in this chapter which have partly been published in reference [215].

An executable system specification is one of the central targets of system development and, in general, represents the development of an overall system model. With regard to the process of system development, executable system specifications are intended to substitute currently used textual specifications. An ESS is derived from the intended application, i.e. mission, as well as the non-functional requirements and constraints of the SuD.

In contrast to the proposal by Baumann (cf. pp. 32 [30]), the mission model is developed during the previous development step as part of a validated executable requirements specification (VERS). This is because the integrated overall mission model is required as validated input for this step of development and for an automated validation of the overall system model, especially with regard to the information on system and actor interaction, expected performance, data model, customization as well as system and environment configuration model. In general, an executable system specification is defined as follows:

Definition 21 An **Executable System Specification** is a coupled model of intended system architecture, functions and system environment (i.e. system context) for a system under design at a specific level of abstraction. It combines at least the aspects of information flow, material flow and control flow into a single model. Executable specifications can be executed in a simulator for validation purposes. This can be done in conjunction with an associated overall mission model or by forming a virtual prototype based on human machine interface models.

Executable system specifications can be described by a 9-tuple $(A, F, ENV, M, GUI, C, P_S, P_{ENV}, P_C)$ where:

- $A = \{a_1, a_2, \dots, a_n\}, \forall a \in A : a \in \{\text{subsystem, element, resource}\}$ is a finite non-empty set of architecture components
- $F = \{f_1, f_2, \dots, f_m\}$ is a finite non-empty set of function components that are allocated to elements $e \in E, F \rightarrow E$
- $ENV = \{env_1, env_2, \dots, env_o\}$ is a finite non-empty set of system environment components
- $M = \{m_1, m_2, \dots, m_x\}, \forall m \in M : m \in \{\text{actor, non-functional observer}\}$ is a finite set of mission model components
- $GUI = \{gui_1, gui_2, \dots, gui_y\}$ is a finite set of human machine interface model components in the form of graphical user interfaces
- $C = \{c_1, c_2, \dots, c_z\}$ is a finite non-empty set of customization parameter memories
- $P_S =$ is a finite non-empty set of configurable system parameters
- $P_{ENV} =$ is a finite non-empty set of configurable environment parameters
- $P_C =$ is a global parameter containing the overall data model

Figure 4.48 depicts the basic structure of an executable system specification model together with the division of different design aspects to different model components. The interactions between components of an executable specification models are controlled by domain specific scheduling algorithms. By using a multi-domain modeling and simulation tool like MLDesigner, different computation domains can be covered by one model, including discrete event (DE) domain, finite state machines (FSM) domain and synchronous data flow (SDF) domain.

The complex behavior of the overall system under design is not solely defined by its functions alone. Instead, intended system functions are enabled and influenced by specific aspects of system architecture and system context. In order to include design uncertainties in the overall model, bounded value ranges are used for all non-functional design parameters instead of fixed point values. Functional components in conjunction with architectural elements (including functional and non-functional architecture aspects), resources and the dynamic coupling between model components result in a more complete image of the SuD. As a result, the overall model

of an executable system specification is divided into three major submodel aspects that describe intended function, environment and architecture specifications as the result of the overall design process.

Function model components represent software aspects of the SuD that are targeted at the provision of system application or mission specific properties. This means, that functions are used to describe how mission and service models are realized by the SuD in terms of logic. Although not common for the creation of complex system functions, it is also possible to define logical functions that are determined by hardware components, e.g. in terms of logic circuits. In this case, respective functions are still allocated to the functional model. Functions of an intended product that belong to the physical world are always embedded within a system architecture. As a result, functions model components interact with the architecture model and are allocated to specific components of the architecture model (functional allocation).

One of the fundamental design principles within the field of product design and constructional architecture is called “*form follows function*”. It has been described, for instance, by Sullivan, who states, that the design or form of any thing of the physical and metaphysical world is determined by its particular function [344]. Compared to the design of ESS, it may be concluded, that system architecture follows the intended system application or mission, which is realized by functions. Thus, architecture model structure and components represent a wide variety of system properties that are determined in order to directly or indirectly enable system functions in order to achieve specific missions. System architecture is also influenced by system context, e.g. environmental conditions or available installation space as well as available resources. As depicted by Figure 4.48, architecture models describe aspects of available technology as well as technology that is to be developed, including functional and non-functional characteristics together with system structure. Resource components are used as part of the architecture model in order to describe available or expected quantities of limited resources during system operation. Technology that is described as part of the architecture model includes e.g. power supplies, memories, processing units, network architectures, middleware and operating systems to name but a few.

Networks provide a good example for functional and non-functional design aspects of architecture components. While describing abstract physical characteristics like cabling, a network is also determined by specific functionality in the form of protocols and overall network structure, i.e. network topology. Although functional aspects of system architecture also describe logical relations, the function behind is not intended to provide a specific mission or service for actors of the SuD. Instead, architecture functions provide generic functionality that is required in order to enable more specific system functions.

The overall structure of system architecture or parts of it may be implicitly determined by the existence of specific functions and constraints that require a specific overall topology, e.g. functions that provide sensor data or actuator properties at specific places of the overall system. The same applies to physical elements of system architecture that may also be implicitly defined by a function or by experience, e.g. in the case of a video recording function that is most likely to be allocated to some kind of camera device. In other cases, system architecture can be determined by designers with higher degree of freedom, e.g. in the case of a control function that can

be integrated within a monolithic computer architecture or within a decentralized cluster of smart control units. Thus, in addition to our previous conclusion “*architecture follows intended system application*”, it is also necessary to determine the best possible system architecture for the SuD with regard to overall system performance, i.e. an optimized overall design. Thus, it may be stated that “*architecture follows intended system application at optimal overall performance*”. Moreover, architecture is also determined based on experiences of system architects, designers and current design trends.

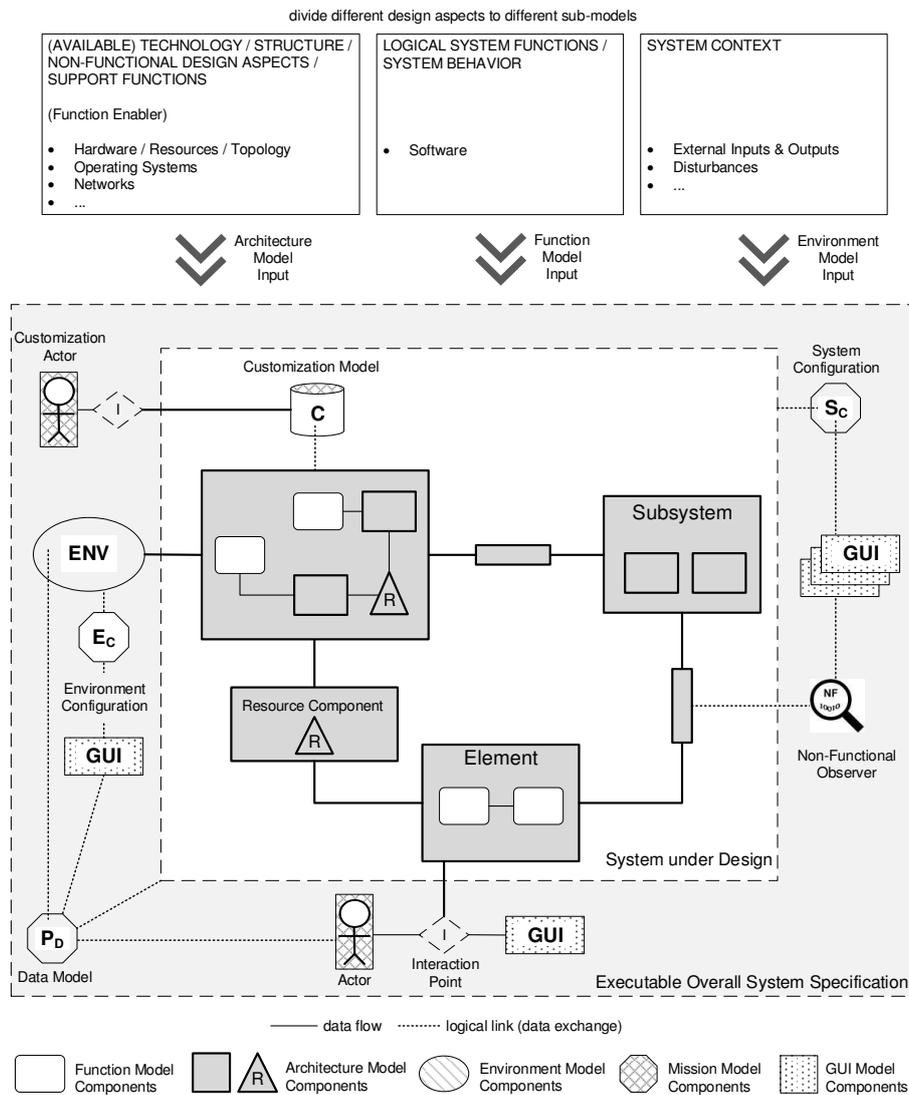


Figure 4.48 – Basic structure of an executable system specification model

At the center of Figure 4.48, the system under design is shown in the form of an ordered arrangement of coupled system architecture and function components that interact mutually. In addition, the customization model that has been developed during the development of a VERS (cf. chapter 4.3.2.2) is shown. By using different parameter values for the customization model, the SuD will provide different functional and non-functional characteristics during model execution. Since customization models consist of data structures with specific sets of parameters, e.g. stored within a database, at least one customization model component exists that is logically linked to one or more components of the function or architecture

model. Customization model components provide interaction points for associated customization actors of the VERS during automated validation. Customization model parameters can also be set manually before simulation, e.g. during interactive validation.

Components of system environment are located in the periphery of the SuD shown in Figure 4.48. In general, system environment has functional as well as non-functional properties and provides inputs for the SuD as well as sinks for SuD outputs. Environment components include external systems, actors at specific system interaction points that have been determined during VERS development (cf. chapter 4.3.2.1) as well as models for system disturbances or external resources. The overall data model of an ESS, which is responsible for any information exchange during overall system execution, includes the associated VERS data model as well as an ESS-specific data model.

Non-functional parameter observer probes that were determined during VERS development (cf. chapter 4.3.3) are linked to global parameters or components of the ESS in order to observe non-functional parameter values, i.e. overall system performance, during model execution. Configuration parameters for system and system environment that have been determined during system and system environment configuration model development (cf. chapter 4.3.4) are linked to the overall system or system environment. They are used in order to define initial system and environment states and characteristics as well as for dynamical state changes during overall model execution. As show in Figure 4.48, graphical user interface (GUI) modules can be used at system interaction points instead of actor modules. Moreover, GUI modules can be integrated and linked with non-functional observer probes or system and system environment configuration models. By using GUI model components that were determined during human machine interface (HMI) concept model development, the ESS can be executed and validated interactively in the form of a virtual prototype (cf. chapter 4.3.6).

4.4.1.1 Function Model Development

In contrast to functional aspects of mission and service models, function model components of executable system specifications (ESS) describe specific tasks and data processing, i.e. behavior, for the system under design (SuD) in terms of what input information is needed at function input ports and how it is used to create an output at function output ports. Thus, functions are used to provide behavioral solutions for requirements from mission and scenario models. Functions are also used to support other functions in achieving a specific task. In general, a function is described as follows, based on the definition for functions by Marwedel et al. [215]:

Definition 22 A *function* f is an abstract entity describing a transfer behavior for a given design problem. It maps a finite set of inputs I with $I = (i_1, \dots, i_n)$, to a finite set of outputs O with $O = (o_1, \dots, o_m)$, under specific conditions C so that $f(I/C) = O$.

Function models can be created architecture independent at functional level without involvement of architectural aspects if required. However, as part of an ESS, function model components require mapping to architecture model elements and resources for

execution. For the example of developing autonomous underwater vehicles, Liebezeit stated, that function and architecture model are coupled and cannot be designed independently [198]. Functions that are used at higher levels of abstraction and that are enabled by other functions are called emerging functions, based on the definition for emerging functions by Marwedel et al. [215]:

Definition 23 An *emerging function* f^* is a high level function that can only be performed by combining function solutions from lower hierarchical system levels, i.e. no single solution can perform such a function. Functions of the system under design at overall system level are normally emerging functions.

Each function is described formally in a way that allows execution on a computer. In the context of this work, functions are modeled in the form of statecharts or finite state machines (FSM) combined with discrete event (DE) block diagrams. Since automata-based models like statecharts or finite state machines can be formally verified and tested, e.g. in terms of completeness, unambiguity and consistency [380], it is possible for system designers to determine design errors like deadlocks or non-deterministic behavior early during development. Other possible ways of determining functions include source code primitives as well as discrete event models. Figure 4.49 depicts the general structure of a function model component of an executable system specification model.

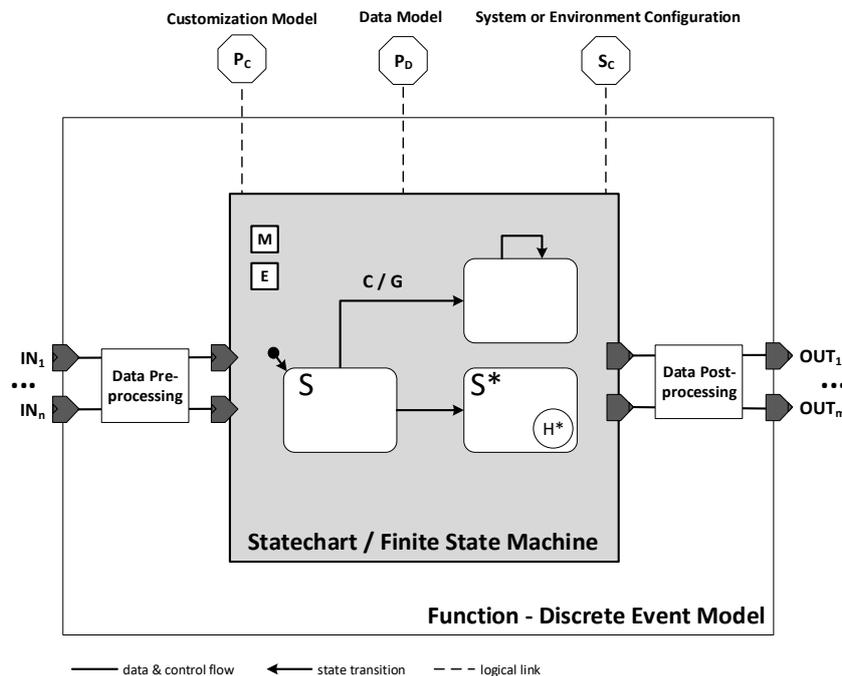


Figure 4.49 – General structure of a function model for executable system specifications

Each function block has finite non-empty set of input ports $IN_{1..n}$ and output ports $OUT_{1..m}$. Both types of ports use data structures that are part of the overall data model. Moreover, function module parameters can be determined that use subsets of the overall data model or that are linked to the customization model, system or system environment model. Incoming as well as outgoing data can be preprocessed

or post-processed, e.g. in order to structure, format, or select data particles. Each function may have one or more statechart modules that are used to describe the overall functional behavior of each function module. Statecharts provide flat states S , hierarchical states S^* with histories H^* , transitions with conditions C and guards G , memories M as well as internal or external events E . States as well as transitions can be used to determine formal functions in terms of an action script language. See references [27], [385], [156], [147] for more details on creating DE as well as statechart or FSM models.

In references [125], [127] and [128] Fischer and Salzwedel extended function modules with a specific internal FSM to represent generic function states like *normal*, *off* and *failed*. Based on the actual state of the FSM, the associated function can be controlled interactively during simulation, e.g. during coverage analysis and test.

4.4.1.2 Architecture Model Development

As described before, system architecture comprises necessary structures, resources as well as functional and non-functional design properties that are required in order to find a feasible solution for a system under design (SuD). According to Schorcht [323], integrated architecture models are a necessary precondition for evaluating performance properties of any system design. Thus, architecture models can be defined as follows, based on the definition for architecture by Marwedel et al. [215]:

Definition 24 *An **architecture model** is a solution to a given design problem. It is a combination of system structure, sets of subsystems, elements, resources and functions that fulfills a specific mission. An architecture is an element of the design space for the system under design.*

At the beginning of system architecture design, a so-called conceptual architecture is formed with the aid of system architects. A conceptual architecture describes a generic structure of subsystems and other physical components called elements. By creating a conceptual architecture, system architects provide the basis for functional allocation and detailed subsystem development. Moreover, conceptual architectures provide a starting point for combined mission and performance validation and evaluation [215], [125], [127] and [128]. Later, during overall design optimization, the conceptual architecture is substituted by an optimized overall system architecture (cf. chapter 4.4.5). Subsystems are architecture models at lower hierarchical levels and are used to decompose a system into task-specific submodels. Elements are entities that represent a technical solution with specific overall structure that combines software aspects, in the form of functions and support functions, and hardware aspects. Support functions can be defined as follows:

Definition 25 *A **support function** f^+ is a special case of a function and is intended to support the tasks that are to be provided by one or more functions. Support functions are typically integral part of architecture and environment models and are closely coupled to the specifics of a technical solution.*

Elements are intended to host and enable sets of functions or support functions with specific performance and are also referred to as execution units. In his work, Baumann uses task-specific names for different system architecture elements, including

partitions, channels and executers [30]. In the context of this work, elements are defined more generically based on the definition for elements by Marwedel et al. [215]:

Definition 26 *An **element** E is a part of an architecture model solution performing at least one function or support function. Each element provides a set of non-functional design parameters in the form of objectives and requires resources for operation. Elements can be arranged into systems, subsystems or higher level elements to achieve the desired behavior and performance for a given design problem.*

Figure 4.50 depicts the relation between functions, support functions and elements. Elements include, but are not limited to, models that represent hardware components, operating systems as well as network protocols. Based on the mission level design approach by Schorcht, elements can be developed in the form of specification models to determine components that need to be developed from scratch as well as in the form of macro models that represent an abstracted version of already available technical solutions or detailed models [323].

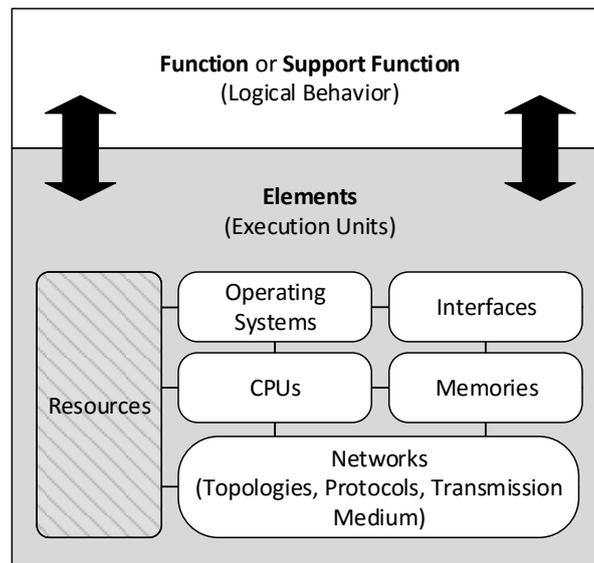


Figure 4.50 – Relation between functions and elements

Resources are always limited and can be divided in two categories. Consumable resources are provided with limited amount and cease to exist once they have been consumed, e.g. a resource for electrical power that is provided by a battery. Allocatable resources are also provided with limited amount, but can be allocated for a specific amount of time and released afterwards, e.g. memory or bandwidth. Resources can be supplied by system architecture elements or entities of system environment, e.g. a resource electrical power that is supplied by a generator unit. In this case, an element may use one resource in order to supply another, i.e. fuel is transformed into electrical power. In the context of this work, a resource can be defined according to Marwedel et al. [215]:

Definition 27 *“A **resource** $r(E)$ is a quantity of a physical entity that is required for operation of an element E .”*

Objectives have been already described in chapter 4.3.3. In terms of architecture models and elements, objectives describe constraints in the form of bounded parameter value ranges that determine the performance of the related technical solution, i.e. an element. Each objective reflects domain specific expert or empirical knowledge on expected, known or measured architecture component performance. Bounded value ranges are used instead of fixed point values in order to include design uncertainty that is specific to each element. In the course of system development and test, estimated objectives for new elements can be substituted with more accurate measurements in order to decrease design uncertainty further. The same update principle applies to resources. At the beginning of design, resources may already be known or are estimated based on experience.

Figure 4.51 depicts the general structure of an element model component of an executable system specification (ESS) model. Each element uses the overall data model of the associated ESS and has an arbitrary set of input and output ports that for receiving and sending data. Elements can be linked to system configuration and customization parameters that can also be linked to other sub-components. In addition, objectives for non-functional design properties can be determined and linked to components of an element if necessary. An elements at level L may include a finite number of functions, support functions, element-specific components, or other elements on lower hierarchical level $<L$. Non-functional observer probes from mission development are used to couple ESS and quality models of the overall mission model (cf. chapter 4.3.3). Mission model actors are connected to elements externally via input and output ports at specific system interaction points (cf. chapter 4.3.2.1). Since each actor has a specific data model, actor interfaces may be used in order to pre-process or post-process data that is to be exchanged between elements and mission model actors.

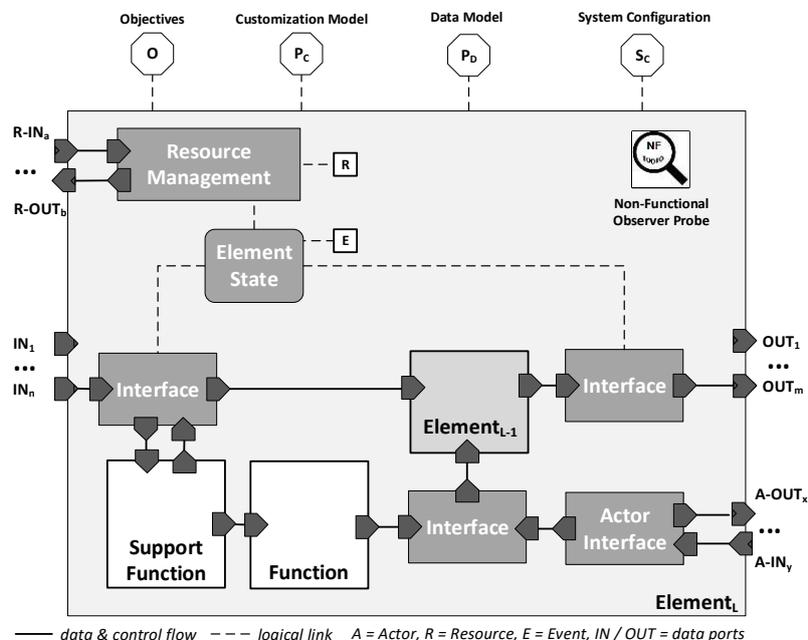


Figure 4.51 – General structure of an element model for executable system specifications

Functions can be coupled with other functions or support functions, as long as both operate on the same data model. Elements use specific interfaces for incoming and

outgoing data. These interfaces operate with a formal data model in order to exchange data between elements and other elements or elements and functions. This mechanism has been introduced by Baumann in order to guarantee consistent architecture and function coupling [30]. In addition, interfaces are logically connected with an internal finite state machine that controls the overall state and operation of each element. It is also possible to couple internal element state and functions in order to describe specific operation modes, e.g. specific test or maintenance modes, that affect overall element behavior [125], [127] and [128].

The internal state of an element can be manipulated by external or internal events E or through coupled resource management modules. If, for instance, a required electrical power resource is not available, the internal state of the associated element may become *unpowered*. As a result, the associated element will cease operation, thus rendering all associated functions inactive. Resource management modules can be linked to internal resources R , e.g. to represent memories, as well as to externally provided resources that are received via input ports, e.g. electrical power. It is also possible for an element to provide resources to external elements or components of system environment via resource management modules. In this case, objectives can be used to determine resource parameters. For more detailed information on the principles of function and architecture allocation and interaction based on a formalized data exchange model please refer to reference [30].

4.4.1.3 Environment Model Development

System environment comprises all components from system context that are in relation with the system under design (SuD) when performing a specific mission. As a result, environment models offer a high degree of freedom during modeling and can provide wide range of functional and non-functional inputs for the SuD, including resources and disturbances. The determination of model components that are part of the system environment is always dependent on the level of abstraction for a specific design problem, i.e. the SuD, and may shift when integrating system models at lower hierarchical level into models at higher levels.

In order to be able to allow coupled execution of all models of an ESS, the environment model needs be consistent with the overall data model. One way to provide a high degree of model interoperability is to use the function and architecture model mechanism for environment models as well. Thus, functions, elements and resources are used together with a formalized data exchange mechanism.

As shown in Figure 4.48, the boundary of the SuD also provides possible system interaction points with external actors, including human and non-human entities. As a result, actors that have been determined during the design of an executable requirements specification (ERS) are part of the environment model as well as non-functional observation probes and environment configuration parameters. Since actors and non-functional observer probes have already been determined during ERS development (cf. chapters 4.3.2.1 and 4.3.3), they can be placed at the appropriate points of the executable system specification (ESS). On the other hand, environment configuration parameters are used to determine specific aspects for the overall environment model or its components.

When using a multi-domain modeling tool like MLDesigner, it is possible to develop environment model components with different domains of computation. This may be useful when integrating continuous behavior with event-based system models.

4.4.2 Automated Specification Validation

An important partial result of this work is the ability to develop the means for system developers that support automated system specification validation. By using executable workflows, i.e. simulation sets, it is possible to unite validated executable requirements specification (VERS) and executable overall system specification (EOSS) to create an executable validation flow (EVF). As a result, it is possible to perform an automated validation process for EOSS that can be repeated after performing further steps of system development. During automated validation, the EOSS model is validated with the VERS model, i.e. the validated overall mission model. In parallel, validation reports are generated that provide detailed information on the overall validation process, including failed functional and non-functional requirements. Automated validation needs to be completed successfully in full at least once in order to find a valid solution for the system under design (SuD) at overall system level, i.e. a validated executable overall system specification (VEOSS). A valid solution is defined as follows:

Definition 28 *A fully valid solution to a given design problem has been determined, if the overall mission and performance that is described within a validated executable requirements specification can be fully attained with the associated executable overall system specification. This includes all functional, non-functional and customization design objectives determined by the validated executable requirements specification.*

However, attainment of only a subset of non-functional requirements, i.e. performance objectives, may also be sufficient. This depends on case-by-case assessments as well as engineering or business decisions. Figure 4.52 depicts the general structure of a top-down oriented executable EVF, i.e. a simulation set. An EVF consists of four different node types. Firstly, executable workflow control nodes which are used to structure the overall control flow. Secondly, a central VERS model that includes an arbitrary number of sequentially arranged mission models. VERS models are the focal point of each EVF since they unite validated driver and evaluation model for the SuD. Different mission model profiles can be determined in order to provide different operational concepts for a SuD, e.g. a set of operational models that use different sets of atomic mission and performance models with associated customization. Thirdly, an EVF includes an EOSS model in the form of a parallel node that interacts with the VERS model during EVF execution.

The details of VERS and EOSS interaction during automated validation have already been covered in chapter 4.3.2 and following chapter. During simulation, mission model nodes, including scenarios, quality objective models and environment configuration models, stimulate certain parts of the EOSS model in order to observe and evaluate related reactions. In order to trace the overall validation status during coupled VERS and EOSS execution, a fourth node type in the form of a parallel global validation evaluation and report module is used. This module interacts

with lower level validation report modules of the VERS in order to provide detailed information on the validity of the current design solution, i.e. the EOSS.

Predefined validation process evaluation and report modules have been developed in the context of this work to provide online and offline validation information for the SuD. Online validation information includes widgets with text and graphics that can be used to trace the status of functional and non-functional requirements during as well as after simulation. Offline validation information is created in the form of structured information that is stored within a file system or database for ex post analysis. Primary tasks of any validation report of an EVF are to enable the evaluation of an EOSS with regard to mission and service model execution and to enable debugging. Questions to be answered with the aid of automatically created validation reports include: “*What was the overall system performance during mission execution?*”, “*Which scenarios were executed as required and with the associated performance?*” and in the case of unsuccessful validation “*What kind of invalid behavior and performance occurred during EVF execution and where?*”.

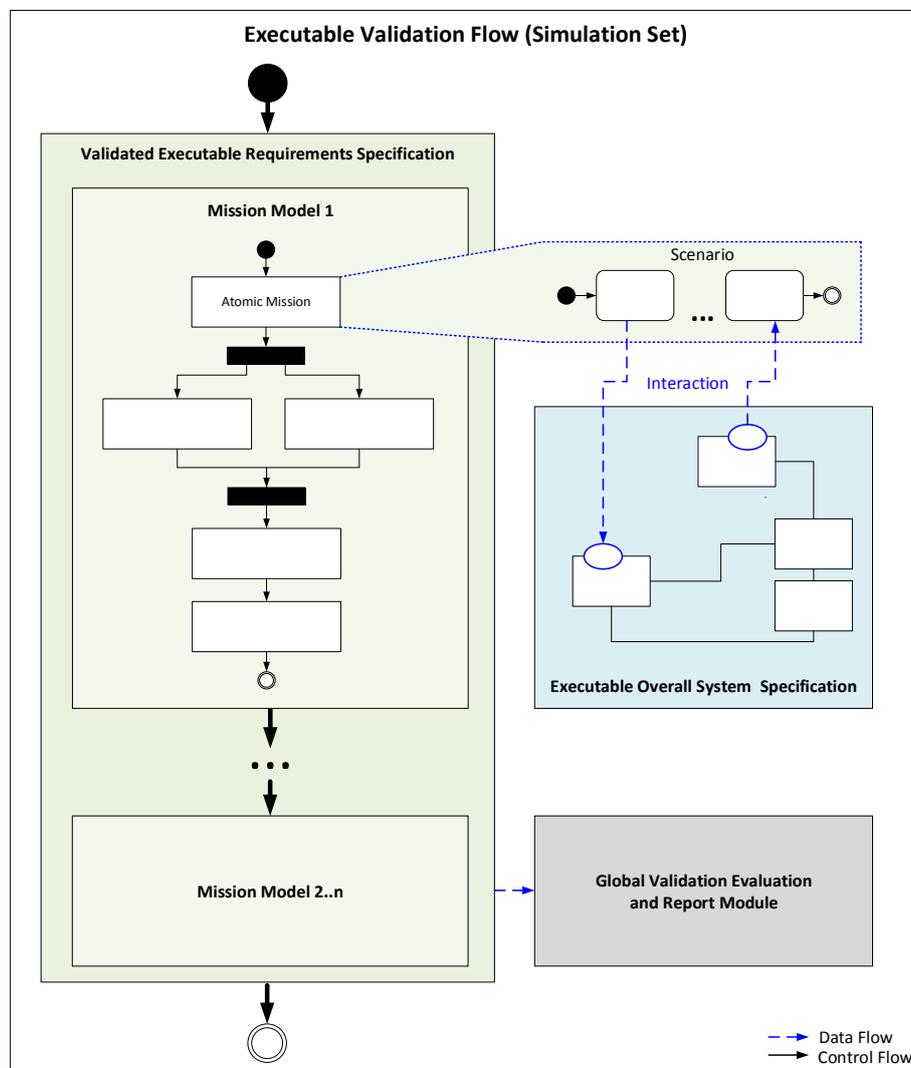


Figure 4.52 – Structure of a top-down oriented executable validation flow, i.e. a simulation set, with a number of sequentially arranged missions as part of a validated executable requirements specification model interacting with a parallel executable overall system specification model and a global validation report module

Each of the developed modules provides a set of basic functions and information that are used in order to trace validation status information. Because of their modular structure, all modules can be customized, e.g. in order to include additional information, to structure information differently or to provide different designs for online and offline reports. Figure 4.53 depicts four different modules that have been created for the purpose of validation report generation. At the bottom of Figure 4.53, two different validation information report modules are shown. Instances of these modules are used like data probes in order to gather validation information for specific parts of mission models during simulation. This information is sent to an instance of a global validation report module which is shown in two variants A and B at the top of Figure 4.53. As the central element of validation report generation during EVF execution, a global validation report module accumulates all information provided by lower level validation report modules in order to determine an overall validation status (validated or failed) as well as a global validation report for the SuD. In order to exchange information between validation information data probes and global validation report module, a task-specific data structure has been determined.

Table 4.2 shows the validation data structure that is used by all validation report modules. The members of the first eleven rows of type string and integer are used to store information on the source of a related validation information that is exchanged between validation report modules. This includes information on the current SuD, as well as associated mission, atomic mission, quality objective, service and scenario models.

Member	Data Type	Annotation
System Name	Root.String	-
System ID	Root.Integer	-
Mission Label	Root.String	-
Mission ID	Root.Integer	-
Atomic Mission Label	Root.String	-
Atomic Mission ID	Root.Integer	-
Quality Objective	Root.String	-
Service Label	Root.String	-
Service ID	Root.Integer	-
Scenario Label	Root.String	-
Scenario ID	Root.Integer	-
Requirement Type	Root.ENUM.Requirement_Type	{Functional, Non-Functional, Mixed}
Quality Objectives	Root.Vector.Objectives_List	Vector of Objectives
Information	Root.String	-
Validation Status	Root.ENUM.Validation_Status	{Validated, Failed}

Table 4.2 – Validation data structure

Enumerated type member *Requirement Type* is used to differentiate between requirement types (functional, non-functional and mixed). Information on performance-related requirement parameters are stored within member *Quality Objectives*, which is a vector of type objective. Any other validation information can be written into string type member *Information*. Finally, member *Validation Status* is used to determine the overall status of related requirements (validated or failed).

At the bottom left-hand side of Figure 4.53, module type *Report_Validation_Info* is shown. This type of module is included with predefined parameter configurations

within all quality objective model modules that have been developed in the context of this work (cf. chapter 4.3.3). *Report_Validation_Info* has three input ports and the following set of configurable parameters: *System_Name* and *System_ID* are used to determine the associated system under design, while *Mission_Label* and *Mission_ID* are used to determine the associated mission. Parameters *Service_Label* and *Service_ID* are used in case the associated quality model is part of a service model. Port *Quality_Objectives* is used to receive and store information on quality objectives of the associated model during simulation while port *Information* is used to receive and store any additional information in the form of structured text. Port *Control* receives integer numbers and is triggered and evaluated at the end of quality model execution. It is used to determine the overall validation status (1 = validated or 0 = failed) of the associated model. By default, quality objective validation report modules provide information on the specifics of each non-functional requirement during automated validation, including lower and upper parameter boundaries as well as an overall mean value. In addition, textual information is provided on other properties of each non-functional requirement.

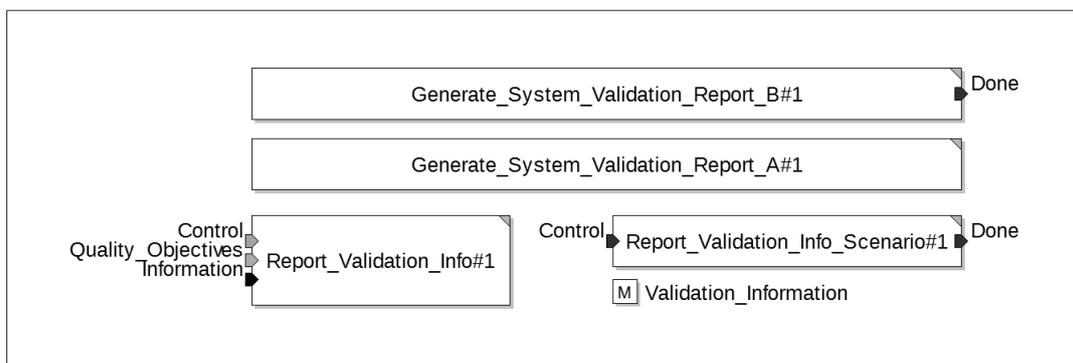


Figure 4.53 – Different automated validation process evaluation and report modules

At the right-hand side of Figure 4.53, module *Report_Validation_Info_Scenario* is shown. This module is used as mandatory part of scenario modules (cf. chapter 4.3.5.2). *Report_Validation_Info_Scenario* has one input and one output port, an externally linked memory as well as the following set of configurable parameters: *System_Name* and *System_ID* are used to determine the associated system under design, *Mission_Label* and *Mission_ID* are used to determine the associated mission and *Atomic_Mission_Label* and *Atomic_Mission_ID* are used to determine the associated atomic mission model. Parameters *Service_Label* and *Service_ID* are used to determine the associated service model while *Scenario_Label* and *Scenario_ID* are used to determine the associated scenario model. As described in chapter 4.3.5.2, an external memory module of type validation data structure is used in order to exchange validation information between scenario flowchart and validation report module. During scenario execution, this information includes functional as well as non-functional design aspects. In the case of invalid behavior or performances, scenario flowcharts provide information in order to trace and resolve related design issues, including information on performances, violated preconditions, postconditions and stimuli responses of the EOSS. Each time parameters of the external memory are changed, the module evaluates all changes made and provides information to the associated global validation report module. The input port of the module is used to receive integer-based tokens that are used to determine the validation status (1 =

validated or 0 = failed) of the associated scenario. At the output port, control flow tokens are created in order to indicate the termination of scenario execution.

At the top of Figure 4.53, two versions of a global validation report module are shown. Both versions of this type of module are executed as a parallel executable workflow node. The only difference between both modules is that the module at the top provides an output port for control flow tokens. This is useful for integrating global validation report modules within a different control flow structure, e.g. to synchronize termination of VERS and validation report module execution. By default, global validation report modules have the following set of configurable parameters:

- *Authors*: Used to determine individuals that are responsible for validation.
- *Document Title*: Used to determine a validation report title.
- *File Append Mode*: Set to *false* or *true*. Used to create a new validation file or to append information to an already existing validation report.
- *File Name*: Used to determine name, type and location for validation report files. By default, a structured plain text file is created.
- *Reference Validation File*: Used to determine a reference validation report file for data comparison after validation.
- *Show File After Simulation*: Set to *false* or *true*. Used to determine if the generated validation report is opened after simulation.
- *Show Reference Comparison*: Set to *false* or *true*. Used to determine if the generated validation report is compared to a reference file after simulation.
- *External Report Program*: Specifies an external program that is used to show validation report files after simulation. By default, a simple text editor is used (*MS Notepad*).
- *External Comparison Program*: Used to determine a tool for data comparison after validation. By default, the free software *WinMerge* is used.
- *System Name*: Used to determine the current system under design.
- *Validation Run*: Used to determine a sequential number for the current validation run.

At simulation start-up, model checking is performed for all validation report modules in order to ensure successful report generation. Validation information is collected and evaluated as long as the associated VERS model is executed. At the end of EVF execution, the global validation report is finalized and stored. By default, global validation report modules create a validation report in the form of a structured text file or as character separated values (CSV) file. The latter can be used for evaluation in conjunction with spreadsheet software. If parameter *Show File After Simulation* has been set to *true*, generated validation reports are opened and displayed after EVF execution as illustrated in Figure 4.54.

By default, validation reports generated with the global validation report module include information on SuD description, document title, validation run number, persons responsible for validation or specification authors, date and time of generation and overall validation status. Moreover, information on functional, non-functional and validation status information including failed conditions and constraints is provided for each mission model requirement.

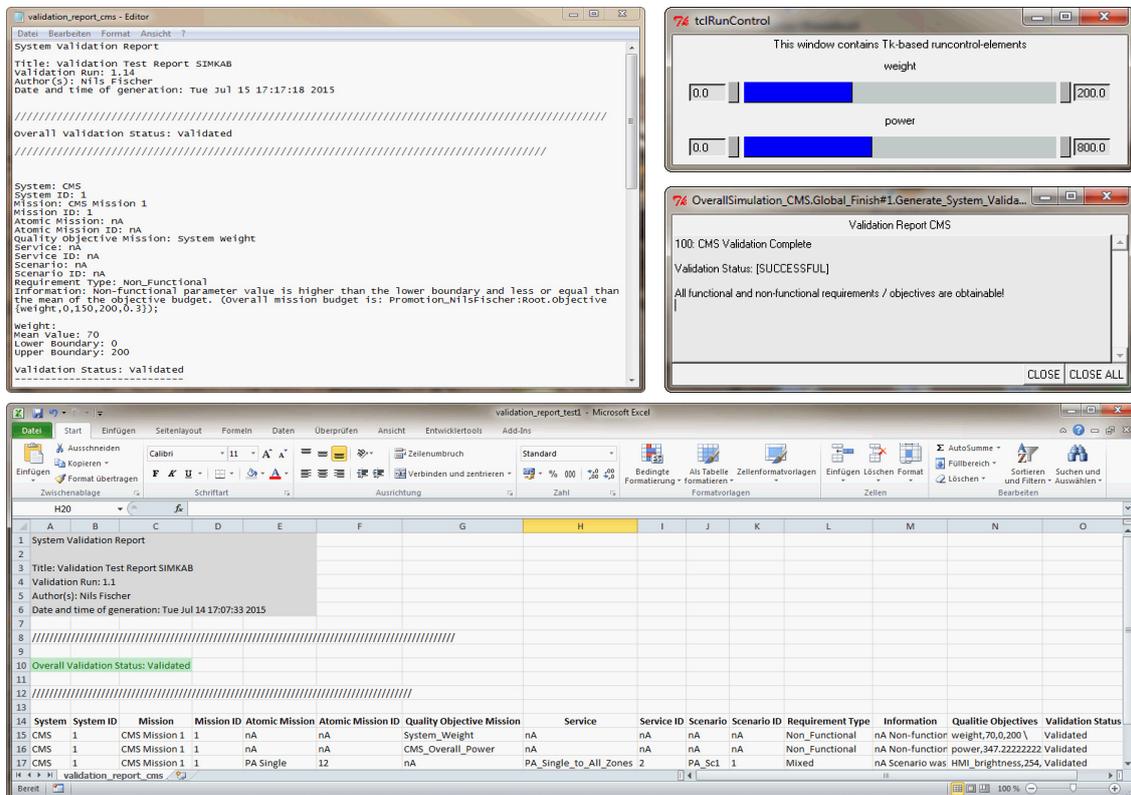


Figure 4.54 – Validation report examples with text (top left) and spreadsheet format (bottom) after automated specification validation together with two online information widgets (top right)

A customizable set of widgets is shown during simulation that provides online information for each validation process. By default, a text widget is generated that shows the overall validation status and, in the case of unsuccessful validation, information on failed requirements and performance objectives. In addition, a bar graph is shown for each quality objective model of the VERS. Figure 4.54 shows examples for text and bar graph widgets that are shown during EVF execution.

Another feature included in global validation report modules is the integration of an automatically executed validation report comparison. By using already existing validation reports from previous successful or unsuccessful validation runs, currently created reports can be efficiently evaluated for changes. This allows for quick re-validation of an EOSS, e.g. after specification updates. In order to activate validation report comparison after EVF execution, and external program needs to be specified with parameter *External Comparison Program* and parameter *Show Reference Comparison* needs to be set to *true*. Figure 4.55 depicts an example for validation report comparison with the free software *WinMerge*. Texts of both files are shown that include highlighted passages and position information in order to indicate changes.

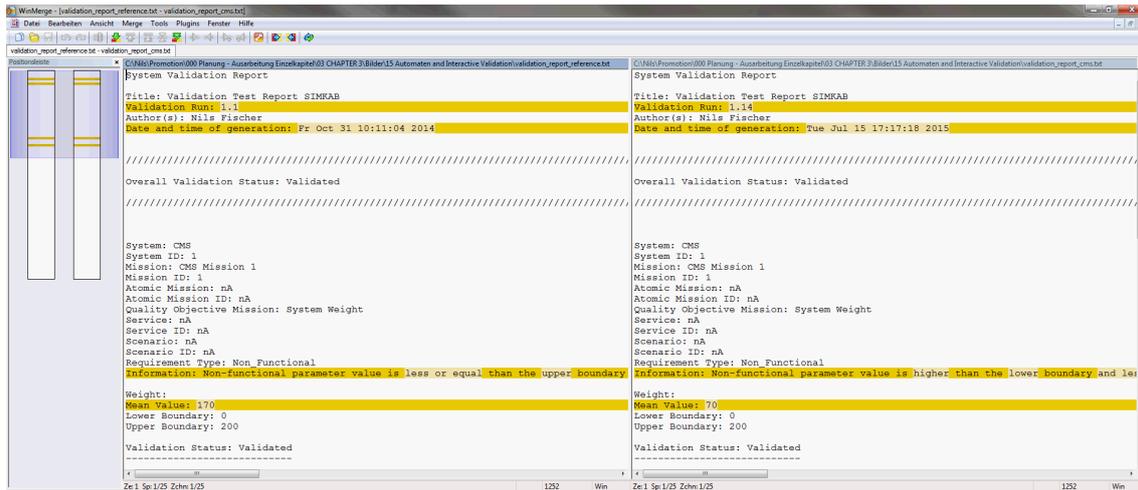


Figure 4.55 – Validation report comparison example with the free software *WinMerge* that highlights changes in currently generated validation reports after automated validation

An EVF can also be created with sets of sub-mission, single service or scenario models in order to validate only partial aspects of overall system design, e.g. for subsystems or components (unit validation / test). In this case, system developers need to ensure that all necessary function, architecture and environment model components are used to create a partial executable system specification that is validated with specific model subsets of a VERS. This approach is useful during detailed development in order to provide unit or subsystem validation capabilities with less complex overall system model structure and reduced simulation efforts before final virtual integration and overall system re-validation.

4.4.3 Towards Automated Reliability Analysis

In addition to executable validation flows, simulation sets could also be used for performing automated reliability analyses of the system under design (SuD). For instance, to perform model-based reliability estimation as shown by Ulrey et al. [230] and [353] for the development of flight control systems (FCS). In their work, Ulrey et al. used a set of two different models in order to perform reliability estimation. An executive model and a behavioral model. During simulation, the executive model is used to perform experiments to deduce properties of the SuD, i.e. the behavioral model, by stimulation and observation of model reactions [230]. This approach of stimulation and observation is similar to the principle of automated validation as described in the previous chapter.

As described in chapter 4.4.1.2, elements of architecture models may contain an internal finite state machine (FSM) that represents their operational status. By default, four different element states are provided: normal, failed, off and unpowered. The operational state of an element can be accessed and changed by using a unique identifier for each element and can be used e.g. to adjust behavior and performance characteristics of elements during simulation. The same principle can be used for function modules or entire subsystems. In references [125], [127], and [128], a graphical simulation user interface was developed in order to monitor and change the statuses of subsystems, elements and functions. For instance, to evaluate the impact of system component failures in the current system design. Moreover,

it is possible to develop failure inducing modules with randomized behavior when developing function and element modules.

By using global accessible operational state FSMs for subsystems, elements and functions or actor modules that are intended to be used for atomic mission model development (cf. chapter 4.3.2), it is possible to link two executable models for reliability estimation, e.g. as described by Ulrey et al. [230] and [353]. Figure 4.56 depicts the general principle of reliability estimation with simulation sets. An executive model is coupled with an executable overall system specification (EOSS) model, which represents the behavioral model used by Ulrey et al. [230]. During simulation, the executive model acts as analysis system for the EOSS. It is used e.g. to inject failures (stimuli) and observe EOSS reactions. A third model is introduced to generate a reliability-related evaluation report during simulation, e.g. to determine the effects observed during simulation. The inclusion of reliability estimation as part of minimum risk model-based development approach with automated validation is currently investigated by Marwedel [216].

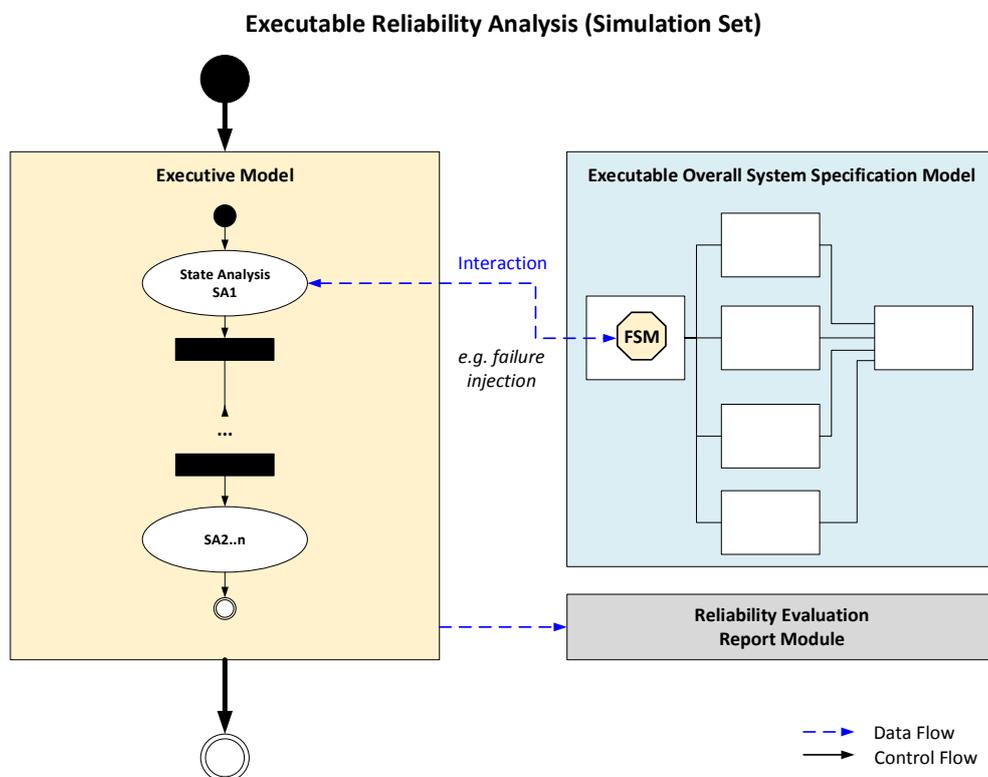


Figure 4.56 – Structure of a simulation set for automated reliability analysis

4.4.4 Interactive Specification Validation, Test and Training

The process of interactive specification validation is a manual validation technique. It utilizes executable overall system specification (EOSS) models or submodels of an EOSS in conjunction with human machine interface (HMI) concept models that were created during executable requirements specification (ERS) development (cf. chapter 4.3.6). The combination of EOSS and HMI concept model is called a virtual prototype (VP).

Most HMI concept model components are created in the form of executable graphical user interface (GUI) models in order to provide users with interactive simulation

observation and control capabilities. GUI models are shown and operated by using different peripheral computer devices during simulation, including keyboards, microphones, displays and speakers. In general, graphical and other user interfaces for VP operation can be used to stimulate inputs for the system under design (SuD) and to observe overall behavior and performance properties. Thus, it is possible to use any task-specific input/output (I/O) device for VP operation. In addition, system as well as system environment configurations can be changed interactively with GUIs during VP execution. By using different customization model (CM) parameter configurations, it is also possible to evaluate the impact of customization on the overall system.

In later stages of system development, virtual prototypes can be coupled with additional external hardware (HW) and software (SW) in order to perform hardware-in-the-loop (HIL) and software-in-the-loop (SIL) simulations. In chapter 4.3.6 it is described how HMI concept models in the form of GUI components are coupled with EOSS models. Figure 4.48 of chapter 4.4.1 shows different EOSS integration and application possibilities for GUI model components from HMI concept model development. However, HIL and SIL simulations require substitution of hardware or software related EOSS model components by external hardware- and software-specific interfaces that do either already exist or need to be developed. All interfaces need to be compliant with the data model of the associated hardware or software component of the EOSS.

Main purpose of a virtual prototype is to perform interactive validation studies of the overall system with different user groups and stakeholders based on human senses and actuators as depicted in Figure 4.57. As described in chapter 4.2, humans tend to easier understand and evaluate information e.g. in order to solve design problems when using imagery. Thus, it may be concluded that one advantage of performing interactive validation along with automated validation is that virtual prototypes provide capabilities for direct feedback of stakeholders on the overall design during simulation. Another advantage is that VP users do not necessarily need engineering skills or detailed understanding of formal methods to be able to operate and evaluate VPs. Interactive validation feedback may induce changes in requirements or the development of new requirements based on instincts or “*gut feeling*” rather than formal design decisions on paper.

Since the stages of requirement elicitation, analysis and validation are characterized by subjective, fuzzy and creative processes, it is often hard to determine a fixed set of system requirements that does not change in the course of overall system development [130]and [129]. By being able to provide a virtual prototype for a SuD that is already executable during system design, changes in requirements and associated system specifications that result from human sentiments and perceptions can be determined early during development. As a result, overall design uncertainty can be decreased even further. Changes determined during interactive validation are fed back to ERS and EOSS development and are included during later automated validation activities. Virtual prototypes are not limited to one specific task and can be used for a wide range of applications. The following list provides an overview of different applications for VPs:

- Manual performed validation of executable specifications against (top level) functional and non-functional requirements during virtual system execution

- Early design and usability analysis (*look and feel*) and system evaluation using human vision and audio sensors as well as human analysis capabilities
- Analysis and validation of different design customization options
- Virtual engineering, including Digital Mock-Ups (DMU), virtual HMI Integration and HMI Prototyping
- Interactive HMI testing and failure analysis
- Early end user training as well as early presentation to customers
- Analysis of alternative application possibilities for the system under design

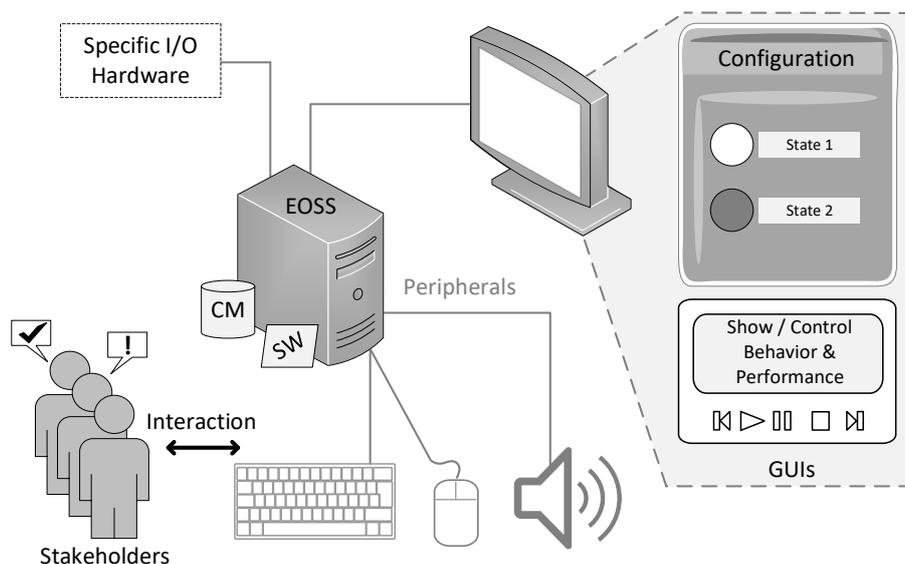


Figure 4.57 – Interactive specification validation with stakeholders and virtual prototypes (composed of executable overall system specification (EOSS) with customization model (CM) and human machine interface (HMI) concept model that includes different graphical user interfaces (GUIs) which are operated with peripheral input/output devices)

Basically, virtual prototypes can be executed and operated intuitively and with no specific sequence of operation. For a more structured validation process, lists that include mission, service or scenario process descriptions could be processed step by step. This could also be done by using use case descriptions or user stories from concept development. Alternatively, mission models of the ERS could be reviewed and recreated directly. It is also possible to encourage different user groups and other stakeholders to operate the virtual prototype independently and ad libitum. The latter is suitable for testing yet unknown scenario processes and combinations. In this case, caution is required by VP users with regard to online changes of system and system environment configurations. If, for instance, system or system environment parameters are configured contradictory or in a way that put the overall system into an abnormal state with regard to other model inputs, the overall behavior and performance of the SuD may become unstable or erroneous.

Validation results from VP execution can be stored in the form of manually compiled spreadsheets or documentation files. Alternatively, it is possible to store interactive

validation process information automatically during execution for ex post analysis, e.g. to reconstruct and analyze user induced system stimuli and associated system responses.

In order to do so, a set of predefined modules has been created as part of this work. Firstly, a global interactive validation report module and secondly, two interactive validation probe (IVP) modules (see Figure 4.59). Figure 4.58 depicts how interactive validation report modules are used as part of a VP. One instance of a global interactive validation report module is placed anywhere within a VP to gather data from IVPs during simulation. This data contains information on time and content of data particles that were exchanged between HMI model components and overall system model during simulation. All information is stored within a character separated file (CSV) file that is used for later analysis. Table 4.3 shows the basic structure of a global interactive validation report files.

At the top of each file, general information is provided, e.g. the name of the SuD, a report title, data and time of generation as well as names of individuals that participated during the validation process. After that, a row with header information is provided that includes identifiers of associated HMI model components that have sent or received data during simulation. All subsequent lines contain cells with information on events that occurred during specific time stamps of overall system execution. Each cell is either empty, i.e. no data has been received by the associated HMI at the respective time stamp, or consists of a data type description and values of particles received at the associated HMI model component during simulation.

General Information (Header) (Lines -k...0)				
	1	2	...	n+1
1	Event Time	HMI ID 1 (in/out)	...	HMI ID n (in/out)
2	t_1	(data type, values) ∨ nil	...	(data type, values) ∨ nil
...				
m	t_{m-1}	(data type, values 1) ∨ nil	...	(data type, values n) ∨ nil

Table 4.3 – Tabular format of global interactive validation report files

IVPs are used like adapters or data probes and are linked between HMI or GUI model components and VP model components in order to gather and record time stamps and contents of incoming data particles. This information is stored within a dedicated local report file and sent to an associated global interactive validation report module instance. As with automated validation process reports, it is also possible to change data storage mechanisms in order to use a database instead of files. For each unidirectional data flow port of an HMI or GUI model component, a separate IVP module instance is used. It is possible to link IVPs with any human actor related HMI or GUI model component. Moreover, IVPs can be linked to GUI model components that are used to depict system quality properties or GUI modules that are used to change system (S_C) or system environment configuration (E_C) parameters during simulation (cf. chapter 4.3.6).

Two different types of IVP modules have been created as depicted in Figure 4.58. While IVP B modules are used passively to gather and record information on exchanged data during simulation only, IVP A modules can also be used actively in

order to reconstruct user induced stimuli from previous simulation sessions during subsequent VP simulations. For this purpose, IVP modules of type A can be configured to work as a driver model for the overall system by using recorded data from previous simulations in terms of system stimulation files. Thus, the resulting simulation of the overall system represents an automated playback of already performed interactive validation runs. A global interactive validation report file is also produced during simulation playback.

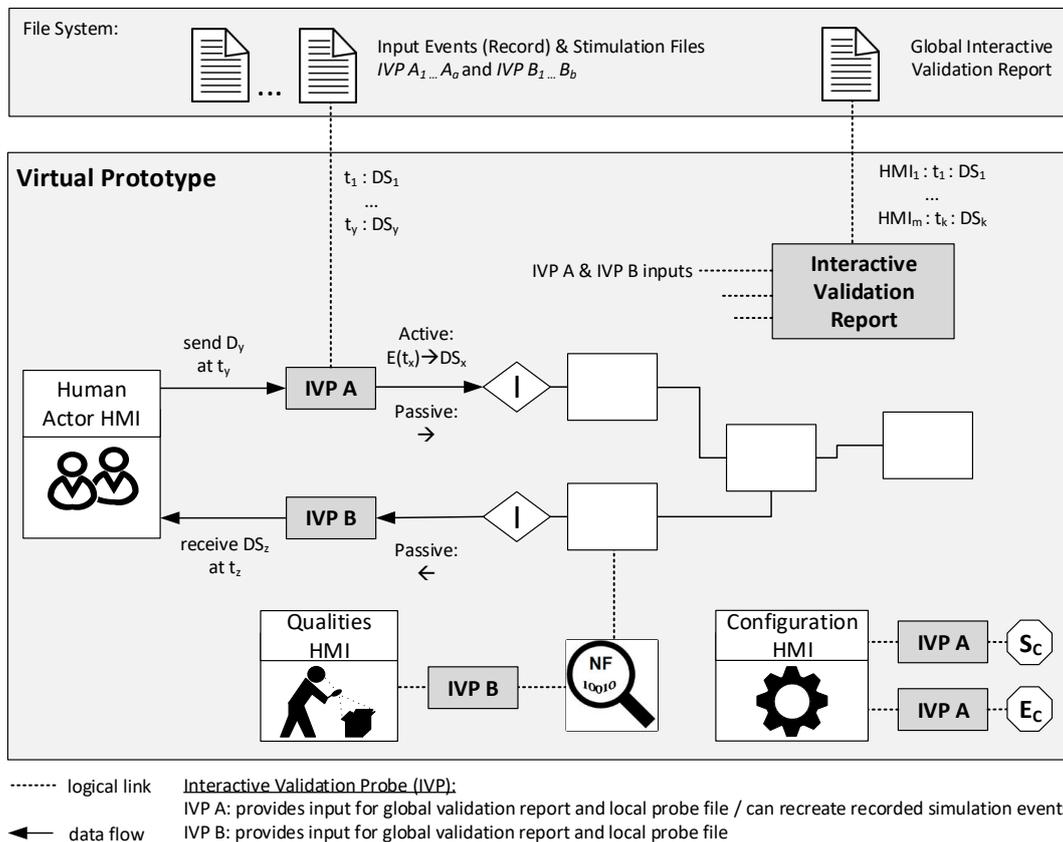


Figure 4.58 – Interactive validation process report module together with interactive validation probes (IVPs) type B (passive) and type A (active or passive) as part of a virtual prototype that can be executed manually or automatically with active IVP A modules

However, using interactive simulation playback based on an IVP driver model is not as sophisticated as automated validation. This is because the EOSS model of a VP can be executed automatically based on IVP driver models but cannot be validated in parallel during overall system execution since no evaluation model exists. This is because evaluation during interactive system validation is performed by stakeholder representatives, i.e. humans. Simulation playback may still be useful, e.g. in order to provide system designers with repeatable and automatically executable validation or test procedures for the SuD that have been determined during interactive system validation by different stakeholder representatives. It is also possible to use simulation playbacks for debugging. Results of consecutive simulation runs can, for instance, be compared by using interactive validation process reports in conjunction with data comparison tools like *WinMerge* as described in the previous chapter.

In general, local IVP report and stimulation files are CSV files with entries separated by tab stops or white spaces. Each line of an IVP A module report file contains a

simulation time stamp together with a set of values for a specific data set DS that is a subset of the overall data model P_D . During simulation with automated playback, IVP A modules substitute specific HMI or GUI model component outputs that are aimed at the overall system model. This is done by creating events at specific time stamps with associated data structures as determined by an associated stimuli file.

Table 4.4 shows the general structure of executable interactive validation probe CSV files in tabular format. The first line of each configuration file must begin with a #-symbol. This marks the beginning of header information for the data structure that is to be created. Each other entry of the first line denotes a specific member name of the respective data structure that is to be created during simulation. In subsequent lines of each file $1..i$, the first row of each line denotes a simulation time stamp t_i with $t_1 \leq t_2 \leq \dots \leq t_i$. All following rows determine specific values for each data set member. During simulation, each IVP A module configured for simulation playback schedules events $E_i(t_i)$ at specific time stamps t for each line of the associated input file. Moreover, each event is linked to the creation of a data particle with specific type and values at the output port of an associated IVP A module.

	1	2	...	n+1
1	#	<i>DS Member Name 1</i>	...	<i>DS Member Name n</i>
2	t_1	value member 1	...	value member n
...				
m	t_{m-1}	value member 1	...	value member n

Table 4.4 – Tabular format for executable interactive validation probe CSV files

In addition to using previously recorded information for repeated overall system simulation, it is also possible to use any text editor or any spreadsheet software in order to create stimulation files based on the format shown in Table 4.4. All basic data types as well as composite data structures and enumerated type data structures can be created by IVP A modules. Numbers, strings and enumerations are written in plain text. Values for composite data structures are encapsulated hierarchically within braces with members separated by commas: $\{member_1, member_2, \dots, member_n\}$. Vectors, i.e. array data structures, are encapsulated in box brackets where all entries are separated by colons. However, the first entry is used to determine the length of the respective vector: $[\text{length}:entry_1: \dots :entry_n]$.

As depicted in Figure 4.59, global interactive validation report modules (left-hand side) have no ports. Five parameters can be used for configuration. Parameter *Document Title* is used to determine a name for each document while *Issue* is used to keep track of generated reports. *Authors* can be used to specify a set of validation participants. Via parameters *File* and *File in Append Mode*, it is possible to choose a location for files that are being created or to append new information to an already existing file. Of course, it is possible to fully adjust and customize validation report modules if required.

As described before, interactive validation probe modules exist in two types. IVP type B, depicted on the bottom right-hand side of Figure 4.59, uses two data ports as well as two configurable parameters. Module type B is passive during simulation, i.e. any data received is directly put on the output port and information on the

data received is sent to the associated global validation report module. Input port *System_Input* is connected to the source of an inbound relation of an associated HMI model component while output port *HMI_Output* is connected to respective input port of an associated HMI model component. Parameter *File Directory* is used to determine a directory for storing local report files for each IVP module while parameter *Probe Info* is used to determine some basic information for creating a global and local report file during simulation. This is done by using a data structure called *Probe DS*. The following data structure members need to be determined for each IVP module. Parameter *Name* is used to determine a name for the associated HMI component while parameter *ID* is used to determine an additional identifier. *Data* is used to determine the data structure type of expected input particles.

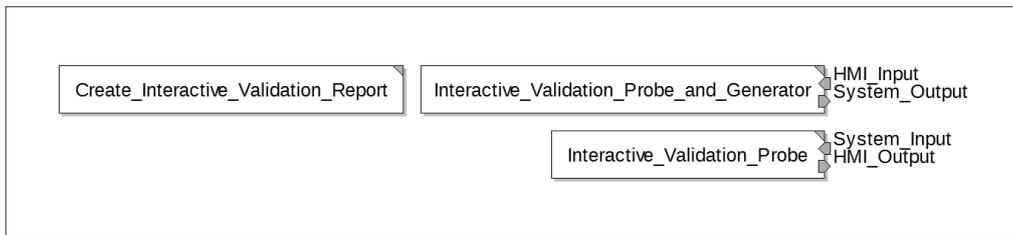


Figure 4.59 – Interactive validation process report module (left) and interactive validation probe modules (right)

IVP type A, depicted on the top right-hand side of Figure 4.59, also uses two data ports and has a set of three configurable parameters. Input port *HMI_Input* is connected to the output port of an associated HMI model component while output port *System_Output* is connected to the target of an outbound relation of an associated HMI model component. Parameters *File Directory* and *Probe Info* are used identically to the same parameters of type B modules. An additional parameter *Playback_Active*, that can be set to either *true* or *false*, is used to configure type A modules in order to operate passively or actively during simulation. If configured for passive operation, type A module behavior is identical to type B modules. IVP A modules that are configured for active operation use report files from previous simulation runs in order to generate events during simulation. With this, it is possible to reproduce previously recorded simulation runs.

4.4.5 Overall System Design Optimization

System design optimization can be performed at different stages of system development. By using validated executable requirements specification (VERS) models and validated executable overall system specification (VEOSS) models, it is possible to shift overall system optimization to early development stages. Thus, an optimal or near optimal solution at overall system level can be determined before commitment of the overall design and the beginning of detailed subsystem and component development. An optimal solution is defined as follows:

Definition 29 The *optimal solution* for a system under design is a fully valid solution for a given design problem that provides an optimal overall system performance or a near-optimal overall system performance with regard to a given objective function or a set of objective functions.

In order to optimize any system design, it is necessary to determine a set of objectives that shall be used to evaluate a given design. This process is strongly dependent on business decisions as well as the intended application of the system under design (SuD). Secondly, it is necessary to determine and analyze parameters of the overall design that can be changed in order to affect overall system performance. Thirdly, an optimization model needs to be developed that is capable to perform design space exploration with regard to a given set of objectives. Finally, it is necessary to combine the validation of each solution of an optimization process with the evaluation of the overall system performance.

Definition 30 *The **design space** is the set of all possible, buildable, and fully valid solutions for a given design problem. A solution is called buildable, if a feasible physical architecture can be implemented and produced later during manufacturing.*

When using VEOSS and VERS models, two general approaches exist to perform overall system optimization. Firstly, the design space can be explored manually in the search for an optimal design. Secondly, the exploration of the overall design space is based on automated techniques, e.g. based on computer simulations. In both cases, different functional allocations, alternative architecture components, system topologies and parameter sets can be used in order to find an optimal solution for the SuD. The two approaches may use automated as well as interactive validation in order to evaluate the current design. Both types of validation provide information on functional and non-functional design properties by means of system specification execution that can be used to evaluate solutions from design space exploration with regard to top-level design objectives. Of course, it is also possible to combine manual and automated design space exploration if necessary.

Manual design optimization is mostly based on experience, engineering judgment or adaptive agile approaches that use cycles of trial and error. The use of manual design optimization may be useful for system architectures with strongly limited design space, systems with a large set of constraints or less complex systems. For this type of optimization, no optimization model needs to be developed and optimization objectives can be evaluated by reviewing the results of automated or interactive specification validation. Manual optimization that is based on engineering judgment may be less prone to the development of unfeasible physical designs. On the other hand, manual optimization can only be used to determine and evaluate specific and selected designs of the overall design space.

In order to be able to systematically explore the overall design space of a SuD, automated optimization techniques with focus on simulation-based methods can be used [313] and [317]. System optimization based on simulation models is performed in many fields of science, engineering and economy. Simulation models comprise different computational domains with different optimization techniques, e.g. stochastic or heuristic optimization for discrete event-based systems optimization [145], [77] and [386]. With regard to executable system specifications, system architects need to be able to determine a system architecture with suitable topology and functional allocation that is optimal for the intended mission and overall performance of a SuD.

For system architecture optimization, especially in the field of aircraft, different simulation approaches have used functional allocation, architecture performance parameters, available resources and system topology in order to determine an optimal

solution for a given design problem. Baumann for example provided methods for automated system model generation [30] that were used by Fischer [124], [316], and [317] in order to determine an optimized avionics system architecture at aircraft level based on automated functional re-allocation. Annighöfer [16] independently used the original approach by Fischer to perform architecture optimization for integrated avionic system architectures. Schulz et al. [325] proposed system architecture optimization for avionics by coupling of computer aided design (CAD) and system design tools. Wichmann et al. [368] developed a system architecture optimization approach with heuristics and run-time reconfiguration of simulation models in order to optimize wireless sensor networks for avionics. The foundation of most architecture optimization approaches is to perform an iterative design space exploration for the SuD with regard to a set of non-functional parameters.

In general, simulative system architecture optimization methods use a set of iterative steps that need to be performed repetitiously in order to find an optimal or Pareto-optimal solution. By using executable workflows as part of an automated design and validation process, it is possible to develop executable architecture optimization approaches for a given design problem. It is also possible to re-use existing model building blocks by adapting already available methods and models, as mentioned above. Figure 4.60 depicts the process of overall system specification optimization process including an executable optimization process, i.e. a simulation set (node with dashed line), which represents the third step of optimization.

At the beginning of the overall optimization process, an optimization model with associated objective and performance evaluation model is developed based on existing VERS and VEOSS models. Optimization models may be based on any optimization method appropriate for a given design problem, including stochastic, gradient-based or heuristic optimization strategies. Objective and performance evaluation models need to be determined with regard to top-level design properties of the SuD.

The actual optimization process is performed during the third step depicted in Figure 4.60, which is represented by an executable workflow, i.e. a simulation set. This simulation set provides a coupled set of simulation models that are executed automatically and iteratively in order to find an optimal solution. At the beginning of the simulation set shown in Figure 4.60, the optimization model is executed in order to perform a design space exploration. In doing so, the optimization model will determine new characteristics and parameter values for the VEOSS model based on the associated optimization objectives. Moreover, the optimization model must be able to cope with possibly unfeasible designs that may be determined during simulation set iteration. All results of this simulation are stored within a persistent memory and are used by the next simulation model in line in order to create a new executable overall system specification (EOSS) with adapted system architecture, including reallocated functions, changes in topology, and parameter adjustment. In other cases, it may be sufficient to adapt parameter values only. As before, changes of the associated EOSS model are stored persistently.

The third step of the simulation set depicted in Figure 4.60 represents the execution an executable validation flow (EVF) (cf. chapter 4.4.2). During this step, the changed EOSS model is validated with the VERS model, including all functional and non-functional requirements. The results of the automated validation process, including overall system performance values, are stored persistently. During the last

simulation of the main optimization loop, this data is used to calculate an objective function for the evaluation of overall system performance (“*Best solution found?*”). In addition, possible results from previous optimization loop runs may be included in the decision-making process (“*Best solution so far?*”).

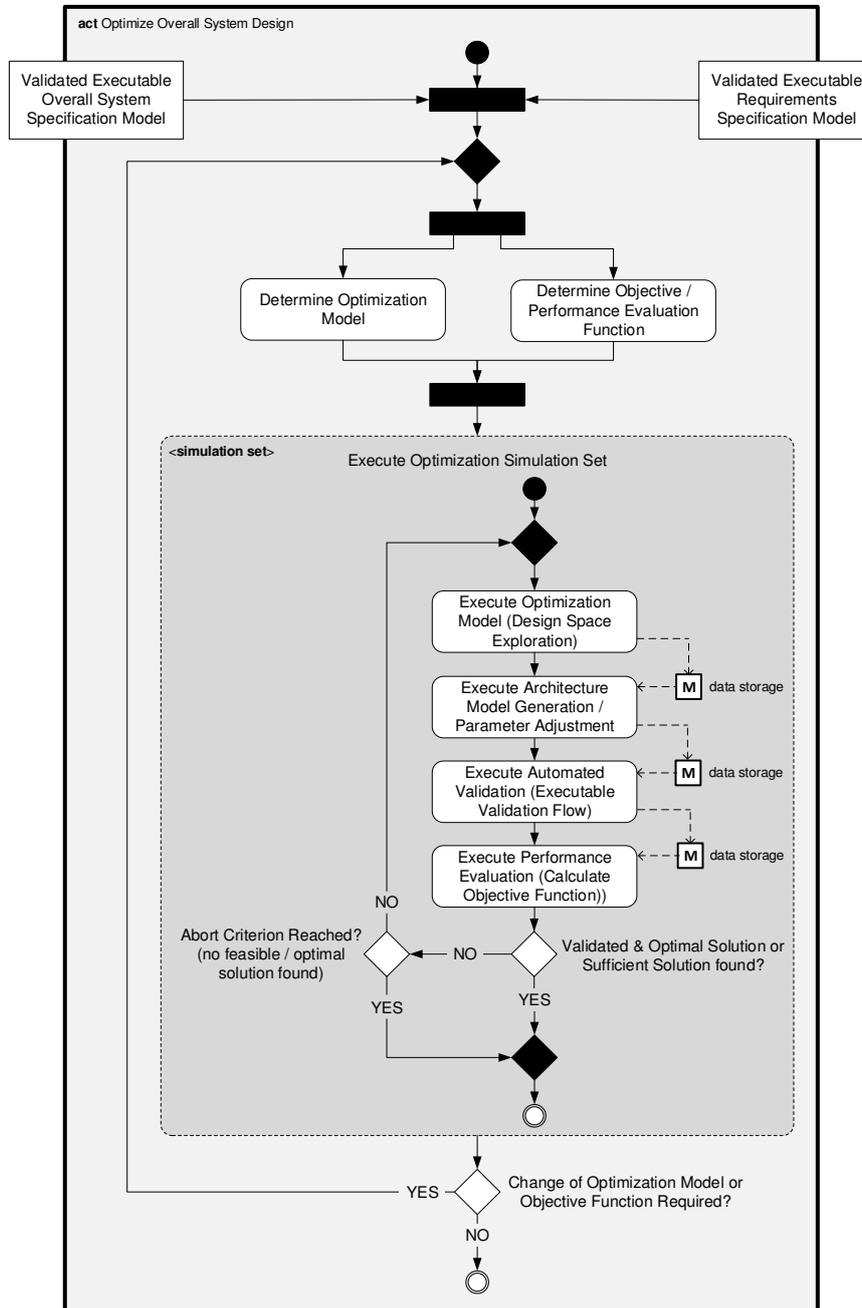


Figure 4.60 – Overall system specification optimization process with executable optimization simulation set (node with dashed line)

The overall simulation set terminates, if the new EOSS design has been validated and an optimal solution or near-optimal solution has been found. The definition of a near-optimal solution for a given SuD depends on the complexity of the overall optimization target and is often characterized by a predetermined number of simulation runs and a bounded range for optimization solution quality. If not successful, it is possible to evaluate a set of abort criteria in order to prevent infinite simulation

loops. After termination of the optimization simulation set, three possible outcomes exist. If the process of overall design optimization was successful, an optimal solution has been found. If not successful, it is necessary to analyze the reasons behind. In case optimization model, objective functions or optimization objectives are flawed and need to be changed, respective models can be reworked before initiating another optimization run. In other cases, it may be necessary to change aspects of the EOSS model or to update the associated VERS model.

4.5 Summary

In this chapter, a concept for creating executable workflows and simulation sets was elaborated. Moreover, a minimum risk model-based engineering (MR-MBSE) approach was developed and elaborated in detail. The developed MR-MBSE approach covers early conceptual design stages of system development as well as executable requirements specification and executable overall system specification development with associated validation activities and concludes with the beginning of detailed system development. An automated validation approach has been developed together with an interactive validation method that is based on virtual prototypes. In addition, the developed MR-MBSE approach includes the description of an overall system architecture optimization strategy based on simulation sets. In order to provide users with the ability to create executable specifications and perform automated and interactive validation activities, a plug-and-play design and validation environment with predefined model components has been created in parallel.

5. Analysis and Validation of the Developed MR-MBSE Method

This chapter is used to evaluate and validate the minimum risk model-based engineering (MR-MBSE) approach that is described in chapter 4. The MR-MBSE design method is validated for design examples of civil aircraft development. Specific focus is set on highly configurable avionics systems, e.g. cabin management systems. For model implementation and the execution of simulations, the system design environment that was developed in this work was used.

5.1 Design of a Basic Cabin Management System

As described in chapter 1.1, growing demands arise for airborne freight and passenger traffic, especially with regard to economic and ecological aspects. Thus, future commercial aircraft require basic methods and technologies to provide the means for the development of efficient and simplified cabin system architectures with high degree of customizability and configurability [97]. Future cabin management systems (CMS) require a system design that perfectly matches the intended application of a CMS, including software that supports airline specific use cases and scenarios. The overall system architecture including software shall be modular, integrated and as lean as possible in order to provide an optimal overall performance. Moreover, a lean system architecture is required to accelerate the process of basic system certification, especially in the context of cabin re-configuration and customization. Central point for enabling the successful development of future CMS is to secure the validity and completeness of specified requirements with regard to all stakeholders [373].

The development of a complete cabin management systems (CMS) for large commercial aircraft carries a high degree of complexity and requires many man-years of development. With regard to existing CMS solutions, current aviation authority requirements as well as cabin technology trends, it is the aim of this example to develop a basic CMS which includes only essential system requirements, i.e. requirements that are regarded to be indispensable for the commercial operation of any civil aircraft [373]. In order to determine a set of essential services, a minimum set was determined from a set of all available service requirements from possible stakeholders.

As a result, a set of seven major atomic missions (AMIS) was determined, including: *Cabin to Ground Service Intercommunication*, *Inter-Aircraft Communication*, *Passenger Address*, *Passenger Calls for Assistance*, *Lavatory Smoke Indication*, *Automated Fasten Seatbelt / Return to Seat Indication* and *Automated No Smoking Indication*. A detailed description of the developed AMIS models is provided in appendix B.1. Within the next chapters, a basic CMS is developed from concept design to system specification.

5.2 CMS Scope, Context and Stakeholders

In a first step during design of a basic cabin management systems (CMS), a central scope of development is defined and system context and stakeholders are modeled in relation to that scope. The definition of a global goal with clear scope, related stakeholders and system context is important in order to be able to determine a complete set of sources for system requirements and to determine relations in-between. If this analysis is incomplete, important CMS requirements may be missing, thus leading to an incomplete design. However, new stakeholders and system environment entities may emerge at later stages of design and need to be analyzed and integrated accordingly. Figure 5.1 depicts an example use case diagram for system context and stakeholder analysis for a basic CMS. In the middle of the diagram, a goal of development (scope) is specified in the form of a global use case called “*Provide Cabin Services*”. This goal describes the essence of the development target for a basic CMS, i.e. to develop a CMS that only provides all essential abilities, functions and services for the safe and efficient operation of commercial aircraft cabin [373].

In the example in Figure 5.1, different external systems interact with the intended CMS. Moreover, CMS are strongly dependent on the specific configuration and customization of an aircraft type. A CMS also bears relation to different physical environments including aircraft cabin, cockpit and, when on ground, ground operation entities. Stakeholders include, but are not limited to, airlines, aircraft manufacturers, (sub-)system providers, cockpit and cabin crew, passengers but also public authorities that issue standards for certification, e.g. the *European Aviation Safety Agency* (EASA). Top-level requirements related to the scope of development are gathered based on needs resulting from system context and stakeholder analysis. This can be done by using market analyses, stakeholder audits, research of regulations and law or the observation of system environment demands. At the end of this process, each requirement is analyzed with regard to the overall goal of development in order to decide what requirements are regarded essential for cabin operations and thus need to be considered in the conceptual design of a basic CMS. This process is elaborated in the following subchapter.

The diagrams that are shown in this section were modeled with the *Toolkit in Open Source for Critical Applications & Systems Development* (Topcased). Topcased can be used as support for conceptual requirements elicitation and analysis. It supports different modeling languages including Unified Modeling Language (UML) and Systems Modeling Language (SysML). When employing use case diagrams to model system context, elements of the context can be modeled in the form of actors. Stakeholders can also be modeled in the form of actors. In order to differentiate between different categories of system context, it is possible to define own categories with related icons. Stakeholders may be differentiated by using different stereotypes

within actor descriptions, e.g. <<user>>. Moreover, stakeholders can be generalized (continuous line with arrow) or shown to be dependent (dashed line with arrow). In Figure 5.1, stakeholders are defined with the basic actor icon while system context is illustrated in the form of so called packages (rectangles). These packages are hierarchical and can be refined, e.g. in the form of block definition diagrams, component diagram or use case diagrams.

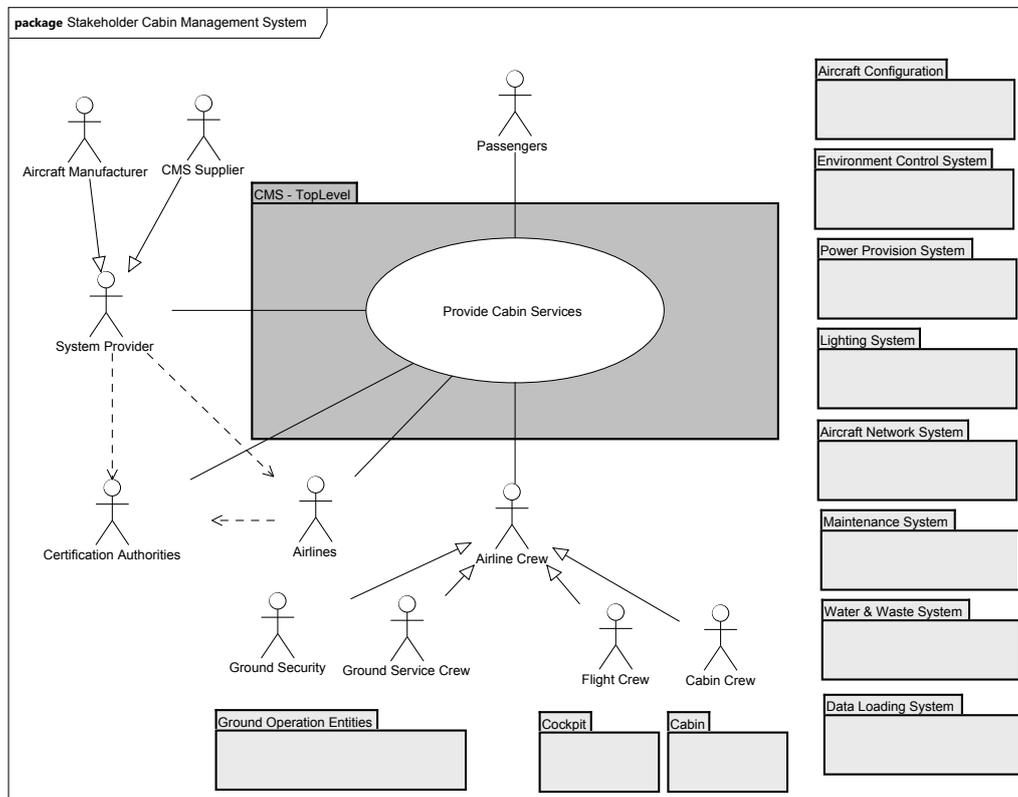


Figure 5.1 – Example use case diagram for a combined system context and stakeholder analysis of a cabin management system (Source: Wiegmann 2014 [373])

5.3 CMS Conceptual Design and Validation

After the determination of a central scope for the development of a basic cabin management system (CMS) and the analysis of system context and stakeholders, a detailed market analysis was performed. This analysis was executed together with the analysis of current trends in aviation technology, certification guidelines as well as safety and security related standards that may affect the development of a potential CMS. Figure 5.2 depicts the first two phases of this process. In the beginning (left-hand side), data of the overall market and stakeholder analysis was compiled and stored within a database (DB).

Currently existing CMS solutions were analyzed with focus on services and performances provided. Moreover, aspects of functional design and system architecture were examined for each system. During this process, data from 24 different CMS solutions were analyzed and stored along with other needs that arise from regulation standards, guidelines and trends in technology. At the end of the process, more than 1000 data sets were recorded. Subsequently, a tool based on spreadsheet software has been developed in order to enable a more detailed and automated analysis of the

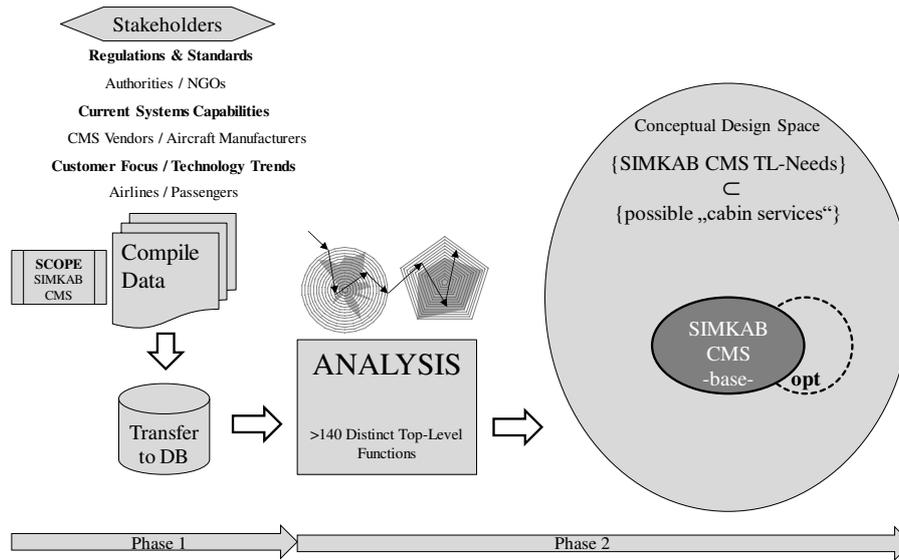


Figure 5.2 – Determining a cut set for a basic cabin management system from a set of top-level needs and possible cabin services ((Source: Fischer and Wiegmann 2013 [130])

overall database with the goal to determine services required by law in combination with services that are most common among existing system solutions. As a result, 140 top-level functions with associated performance criteria were analyzed, evaluated and validated. With this, it was possible to narrow down the overall design space and to determine a minimum set of essential system services as well as optional design properties (right-hand side) [373]. As depicted in Figure 5.3, the global goal of development “Provide Cabin Services” was broken down into six high-level essential use cases that were refined top-down in the following process.

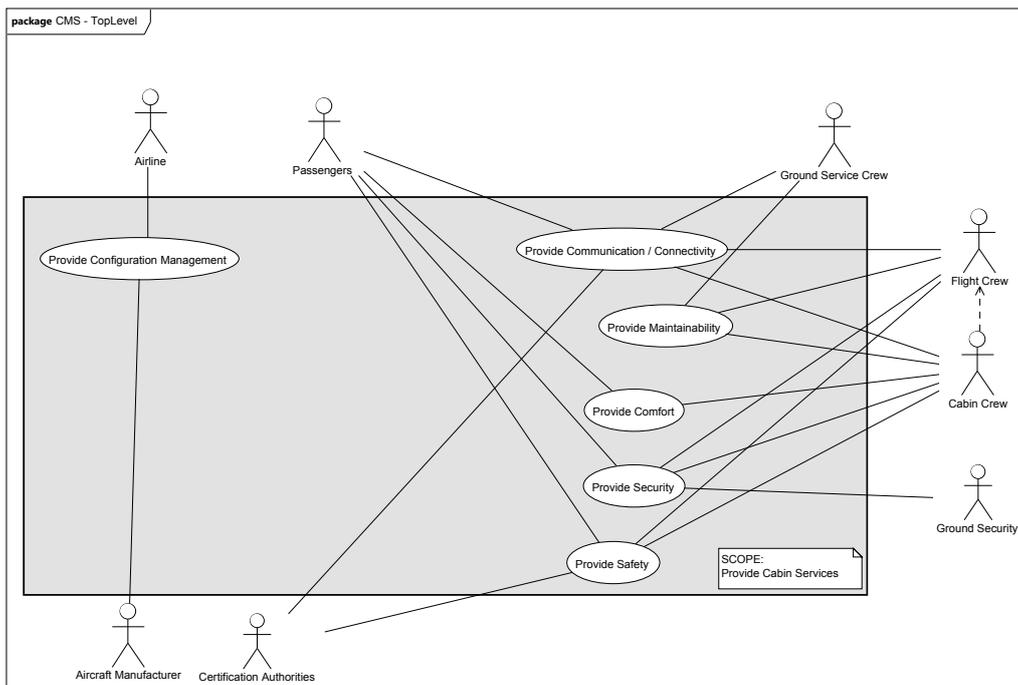


Figure 5.3 – Essential use case diagram example for top-level services of a cabin management system (Source: Wiegmann 2014 [373])

In Figure 5.3, the essential top-level use case “*Provide Security*” is not used as misuse case but to specify a set of aircraft cabin security related needs such as the provision of cabin video monitoring. Figure 5.5 depicts an example by Hintze and God [159] for determining misuse related requirements for cabin management systems. It combines the definition of use cases (white), misuse cases (dark grey) and security use cases (light grey).

In parallel to the creation of essential and concrete use cases, a conceptual design document was created in the form of a mind map. Figure 5.4 depicts part of a hierarchical mind map for a basic CMS that was generated as a result of the process of requirements elicitation and analysis. The mind map in Figure 5.4 was created with the tool *XMind*. Other open source tool alternatives include *FreeMind* and *View Your Mind*. The central node of the mind map shown in Figure 5.4 contains the goal of development (scope) for the SuD. Arranged around the central node of development are nodes that represent the essential top-level use cases depicted in Figure 5.3. Each of these nodes has a finite number of child nodes that represent, for instance, other essential use cases or misuse cases. Again, this is indicated by the preceding phrase “*provide*”. Due to the process of use case refinement, lower level child nodes, e.g. the use case depicted in Figure 5.6, are characterized by higher levels of detail. The conceptual mind map was used consistently during each following stage of development in order to document and trace changes in requirements. In addition, mind map nodes were linked to executable specification models created with *MLDesigner*. At the end of the project, the mind map had ten levels of hierarchy and consisted of more than 1200 nodes in total [373].

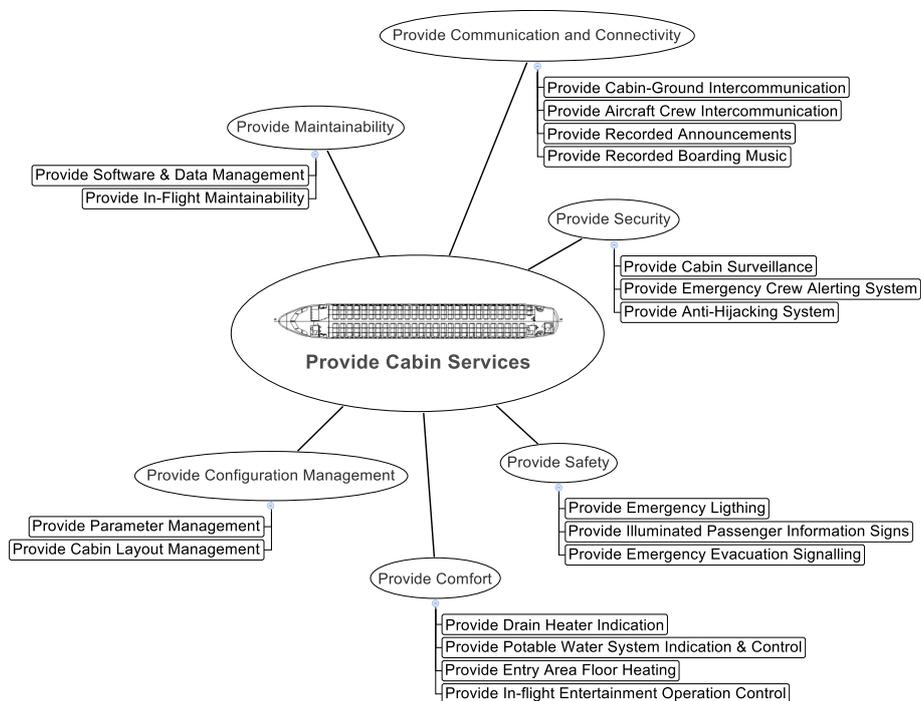


Figure 5.4 – Partial conceptual design mind map example for a cabin management system

In the course of concept development, essential top-level CMS use cases have been refined and validated based on a detailed requirements analysis. During this process, nodes of the mind map were also extended by other diagram types of the Unified Modeling Language (UML) and Systems Modeling Language (SysML) or by linking

external objects such as tables and conceptual graphics, e.g. for early human machine interface (HMI) designs.

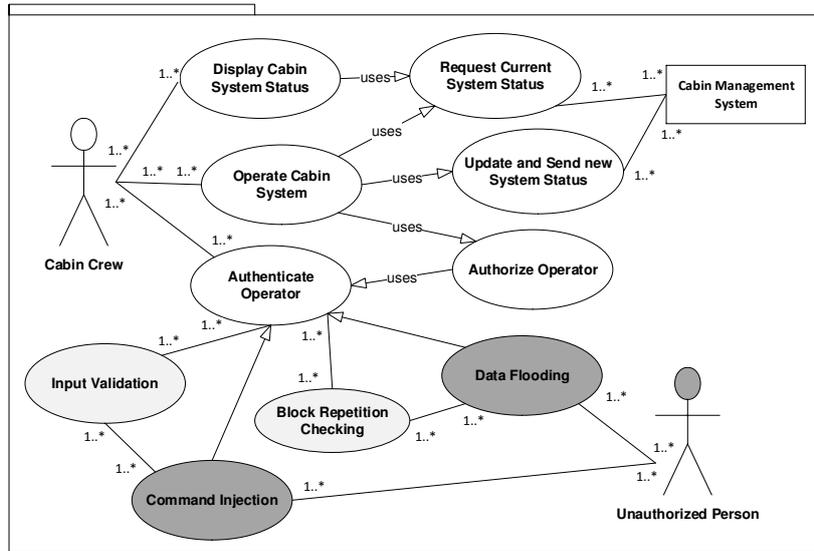


Figure 5.5 – Use cases (white), misuse cases (dark grey) and security use cases (light grey) for cabin management systems (Source: translation of Hintze and God 2012 [159])

Figure 5.6 shows several lower level use cases for a higher level essential use case called “provide call flight attendant functionality” that is part of the essential top-level use case “provide comfort”. It includes more specific use cases as well as continuous use cases that are in relation with different actors.

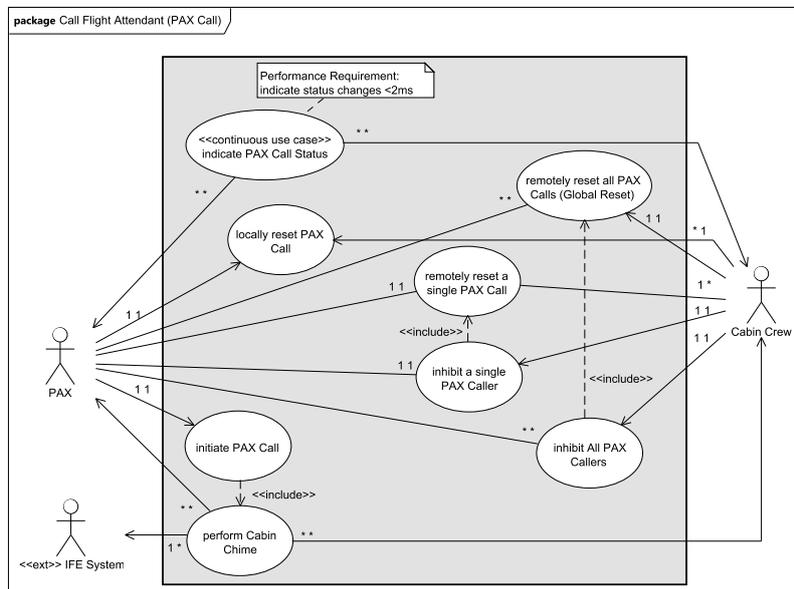


Figure 5.6 – Lower level use cases for the essential use case “provide passenger call for help capabilities” (PAX Call) of a cabin management system

A continuous use case is a special type of use case that continuously provides results to actors and does not require one specific end result [365]. Relations in Figure 5.6 also include cardinality and arrows that indicate if an actor can trigger a use case or participates passively.

In order to prepare the creation of executable requirements specification and service models (cf. chapter 4.3.5), use cases of the conceptual design were also supplemented by conceptual activity and sequence diagrams. Figure 5.7 shows an abstract sequence diagram for a public address (PA) use case of a cabin management system. Instead of explicitly referring to technical implementation details in the form of interaction objects of a PA, e.g. cabin speakers, the PA sequence diagram shown in Figure 5.7 is linked to the abstract actor aircraft passengers instead. In the course of the project, the gradual increase of the degree of formalization beginning with a UML and SysML-based notation proved to be crucial in order to ease the transition from informal top-level needs and concepts down to formal designs, i.e. executable specifications [373].

In addition, the usage of mind maps and graphical models instead of textual documents has proven to be most successful during the process of conceptual design validation. Using a conceptual design based on imagery also provided a solid base for the cooperation between the different stakeholders of the SuD, including system experts and untrained users. This applies to both types of imagery-based modeling, mind maps as well as use case diagrams modeled with UML or SysML [373] and [130]. Feedback provided during concept validation was used to iteratively change the conceptual design before the begin of specification development.

At the end of the validation process, a set of seven major services was determined to be required a CMS with essential services only. A list with descriptions of all validated services that were included in the conceptual design and reasons for why they were included can be found in appendix B.1.

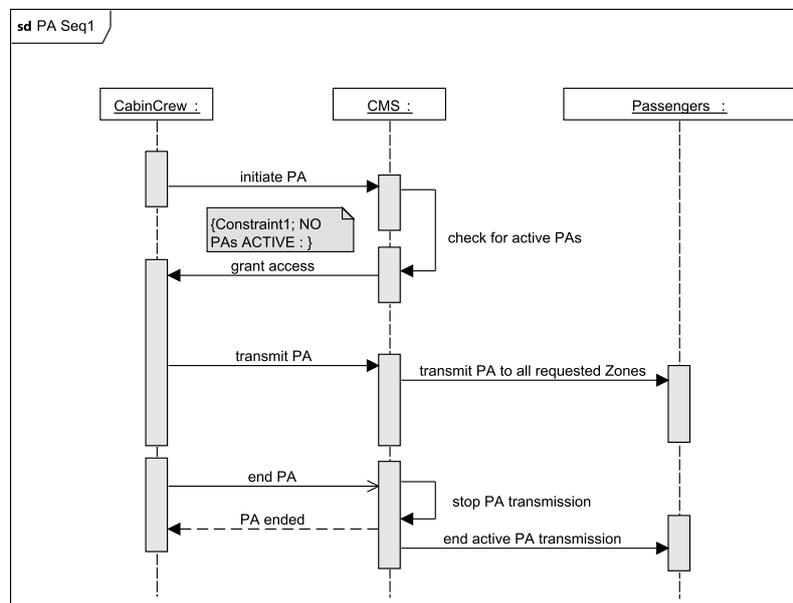


Figure 5.7 – Example sequence diagram to refine an essential use case public address (PA)

5.4 CMS Executable Requirements Specification

In the following chapters, the design process of an executable requirements specification for a basic cabin management system (CMS) is demonstrated by using examples for different top-level requirements. The chosen examples represent major

requirements for CMS, including the safety relevant service “provide public address service” (PA) and the passenger comfort service “provide passenger call for help capabilities” (PAX Call). Moreover, services “provide lavatory smoke indication” (LAV Smoke), “provide passenger notifications” (PNS) and “provide cabin to cabin, cabin to cockpit and cabin to ground intercommunication” (Intercom) are used.

5.4.1 CMS Mission Model Development

With regard to the conceptual design of a cabin management system (CMS), which describes cabin services and system performances to be provided, a typical mission model describes an operational cabin scenario involving different cabin services over time. Missions for CMS are strongly dependent on the operational concept of customers, i.e. airlines. For instance, a mission might begin with boarding and associated services, i.e. use cases, continues over different flight phases and finishes with a set of closing activities on ground. In order to be able to develop a basic CMS, a mission model was developed that represents a typical set of cabin services for most stakeholders (minimum cut set). This was done in collaboration with system experts and other stakeholders. In the course of mission model development, sub-mission models, atomic mission models quality objective models as well as system and system environment configuration models are determined with specific order. This is done by using to information provided from concept development. In addition, experiences of other stakeholders, e.g. system experts, were used in order to determine a valid mission model.

Atomic mission (AMIS) model can be created for each service, i.e. use case, that was determined during concept design, e.g. “*initiate PAX call*” depicted in Figure 5.6. Quality objective (QO) models are derived from top-level quality requirements, including budgets for overall system weight, overall system cost or power consumption to name but a few. In addition to the arrangement of atomic mission models and quality objective models, system and system environment configuration (ENV) models are determined. This is done with regard to the specific requirements of AMIS and QO models. Thus, ENV models need to be configured in a way to enable subsequent models of the overall mission model flow. During mission model execution, environment models of the CMS, i.e. the overall aircraft as well as other systems, need to change their state constantly. At first, an aircraft may be on ground during ground service operations before it proceeds to the runway to lift off. While in flight, different environment parameters may change. For instance, landing gears are retracted and engines provide a specific power output. Moreover, physical conditions like cabin pressure, noise or external lighting conditions may change. Consider an atomic mission model that was developed from an essential use case called “*provide cabin crew to ground service crew communication*”. As part of this use case, an actor of the cabin crew communicates with a member of airport ground personnel in order to manage a variety of tasks, e.g. replenishment of potable water. This can, according to the conceptual requirements of the respective essential use case, only be done when on ground. Thus, as a prerequisite for the successful execution of this specific atomic mission as part of a mission model, system and system environment parameters are altered so that the aircraft is considered on ground and a physical connection to ground personnel has been established.

The arrangement of mission model nodes does not only depend on an underlying operational model of customers alone. Other stakeholders may require specific ar-

rangements of atomic mission models with respect to other top-level requirements, e.g. to determine specific safety or security requirements. As an example in the field of CMS and cabin intercommunication of large commercial aircraft, one requirement of the *European Union Operational Requirements* (EU-OPS) states, that “[...] *the crew member interphone system must operate independently of the public address system [...]*” (cf. EU-OPS 1.690, [4]). Since this requirement of a public authority represents a valid stakeholder requirement for a CMS under design, it is intended to become part of an executable requirements specification. Thus, when determining a mission model as part of an executable requirements specification for a CMS, a mission model might be created that arranges atomic mission models for interphone and public address in parallel. This indicates, that both use cases described within the atomic mission models need to be able to be executed independently and at the same time. In consequence, system designers of later design stages will need to ensure, that the respective executable overall system specification is capable to provide independent operation of interphone and public address functions in order to provide a valid system design.

Figure 5.8 depicts an example mission model for the CMS under development. Starting top-down, the first two mission nodes are quality objective models. All quality objectives are described in the form of bounded ranges instead of point values (cf. chapter 4.3.3). The QO model on the left is used to determine an objective for overall system weight, e.g. in kilogram, while the QO model on the right determines an objective for overall system cost, e.g. in US dollar. Both quality objectives are implemented with static QO models since overall weight and cost of the system under design are not going to change during overall system execution. Another QO model node, in the form of a parallel node, is found to the right of the overall system cost QO model. This node is a detached QO model that determines a dynamical objective for overall system electrical power consumption, e.g. in watts or watt-hours. Energy consumption objectives are typical examples for continuous mission nodes since energy consumption changes over time due to different overall system states and different levels of energy utilization. Therefore, it is crucial to validate dynamical objectives like energy consumption objectives during the entire process of mission model execution.

During the following step, two parallel system environment configuration nodes are used to set the aircraft environment to be on ground with electrical power switched on (aircraft power plant is set active with specific mode). This system configuration acts as a prerequisite for the following mission nodes that are executed in parallel. An atomic mission model on the left-hand side is used to perform a cabin crew to ground service crew intercommunication service. At the same time, atomic mission models for cabin crew to cabin crew intercommunication (middle node) and public address (right-hand side node) are performed. After all three models were executed, atomic mission models PAX Call and LAV Smoke are executed in parallel.

Subsequently, another ENV model is executed. This module is more complex than previous ENV nodes since it is a custom ENV model that was modeled in the form of a statechart. It is used to describe a complete cycle of all possible flight phases of an aircraft during operation. Each reception of a control flow token triggers a change of state with associated configuration changes in system and system environment models. After each change, a decision control node is used to determine

the completion of the overall flight cycle by evaluating control tokens received. If not completed, two atomic mission models are executed in parallel which determine two different automated passenger notification services (PNS).

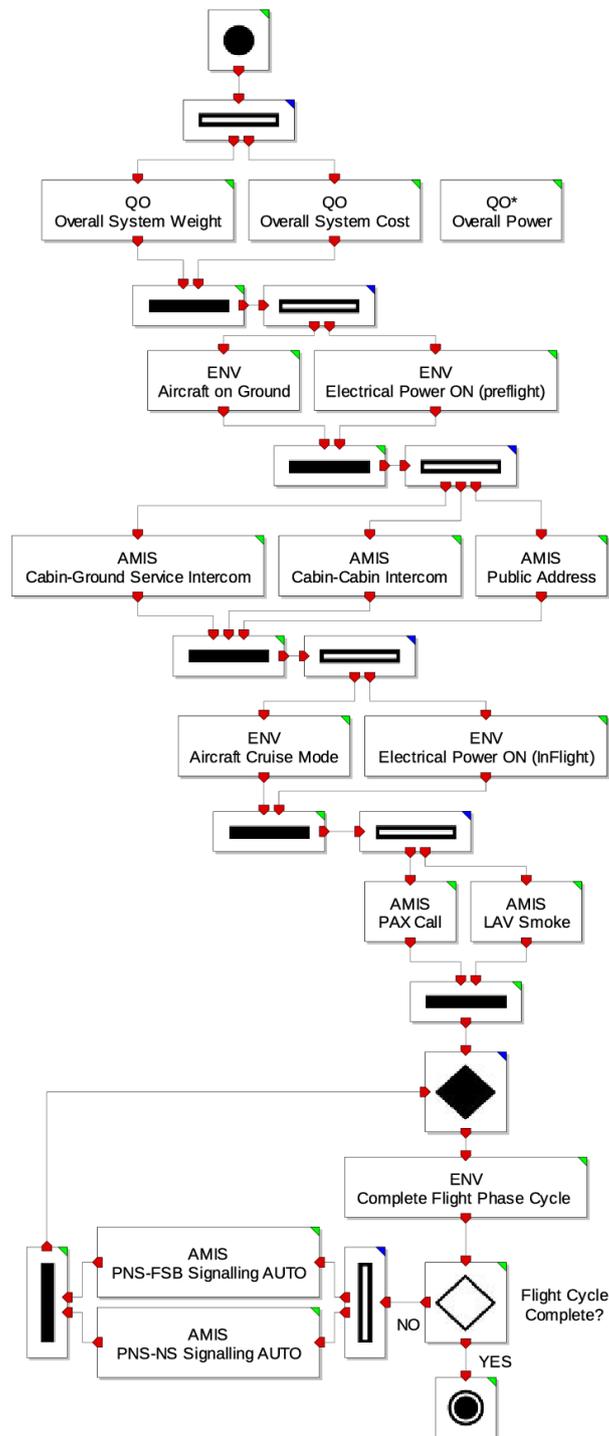


Figure 5.8 – Mission model example for a basic cabin management system

Both atomic mission models depicted at the bottom of Figure 5.8 are derived from the essential high-level use case “*provide status information for passengers*” and depend on the current flight cycle in order to determine different modes of operation. The topmost AMIS model describes a use case for automatic operation of visual

fasten seatbelt (FSB) and return to seat (RTS) indication while the second AMIS model describes an automated no smoking (NS) indication service.

FSB and NS indications require different indications (e.g. on or off) during different flight phases. To validate both services for all associated system environment states, an iterative loop was chosen. Thus, the last part of the mission model shown in Figure 5.8 forms a control flow loop. Both AMIS model examples represent a simplification of the original use cases automatic FSB and NS indication, since both indication services are not exclusively dependent of flight phases alone. This was done for reasons of readability of Figure 5.8. After successful completion of all flight cycles, the overall mission model ceases to execute.

5.4.2 CMS Atomic Mission Model Development

Actor Interaction and Data Model Development

Starting from the developed mission model for a basic cabin management system (CMS), atomic mission (AMIS) models are developed for each top-level use case. Essential and concrete use cases of concept design are directly modeled with MLDesigner by using basic AMIS model components, e.g. actors. In parallel, actor interaction and data models are developed by using data from concept design. As examples for AMIS model development, two essential top-level use cases are modeled in the form of atomic mission models.

Use case example one is called "provide passenger call for help capabilities" or, in short, "*PAX Call*". As part of this use case, a passenger requires assistance to be provided by a cabin attendant, e.g. to request a blanket at his seat. Another form of PAX Call is initiated from inside a lavatory, e.g. to call for personal assistance. A PAX Call can be activated and deactivated by any passenger and shall be indicated to the passenger itself as well as all cabin crew aboard the aircraft, both visually and acoustically. Abstract interaction points are determined that are associated with specific actors and use corresponding data models. In this case, one interaction point is in close proximity to passengers, e.g. as part of an overhead console that allows to activate a call for help request and, in parallel, indicates the active or inactive state of the request, i.e. a human machine interface (HMI). In the course of atomic mission model creation, each actor gets a unique name that indicates its relation to a possible interaction point. Accordingly, a pair of AMIS and ES actors is created with name "*PAX Caller*". This actor is intended as a link to a combined interaction point for triggering PAX Calls and status indication.

In terms of cabin crew interaction points, audible information is provided to cabin personnel, e.g. via a chime at a number of different points of the cabin via cabin speakers or any other audio device. Moreover, a second interaction point is determined that is used to provide visual information of some kind to cabin personnel, e.g. via a cabin information panel. In this case, two cabin crew AMIS and ES actors pairs are determined, one for visual signaling and one for audible signaling. The first pair is named "*Cabin Attendant PAX Call Visual*" while the second pair is named "*Cabin Attendant PAX Call Audible*". Within the annotation parameter of each actor, requirements of associated interaction points are described more closely if required. In the real world, both cabin actors may represent the same person. Thus, it is also possible to determine only one type of cabin actor who is able to

perceive both types of information at once. However, because of spatially divided interaction points, it was chosen to create two separate actor types.

Based on the conceptual design and the intended use case, an associated data model for the atomic mission model is determined. Figure 5.9 depicts a conceptual view of the atomic mission model development for example one. It includes different AMIS actors (bottom) and corresponding ES actors (top), intended data flow (cyan lines), data model (actor adjacent) and interaction points of type HMI (cyan rhombuses).

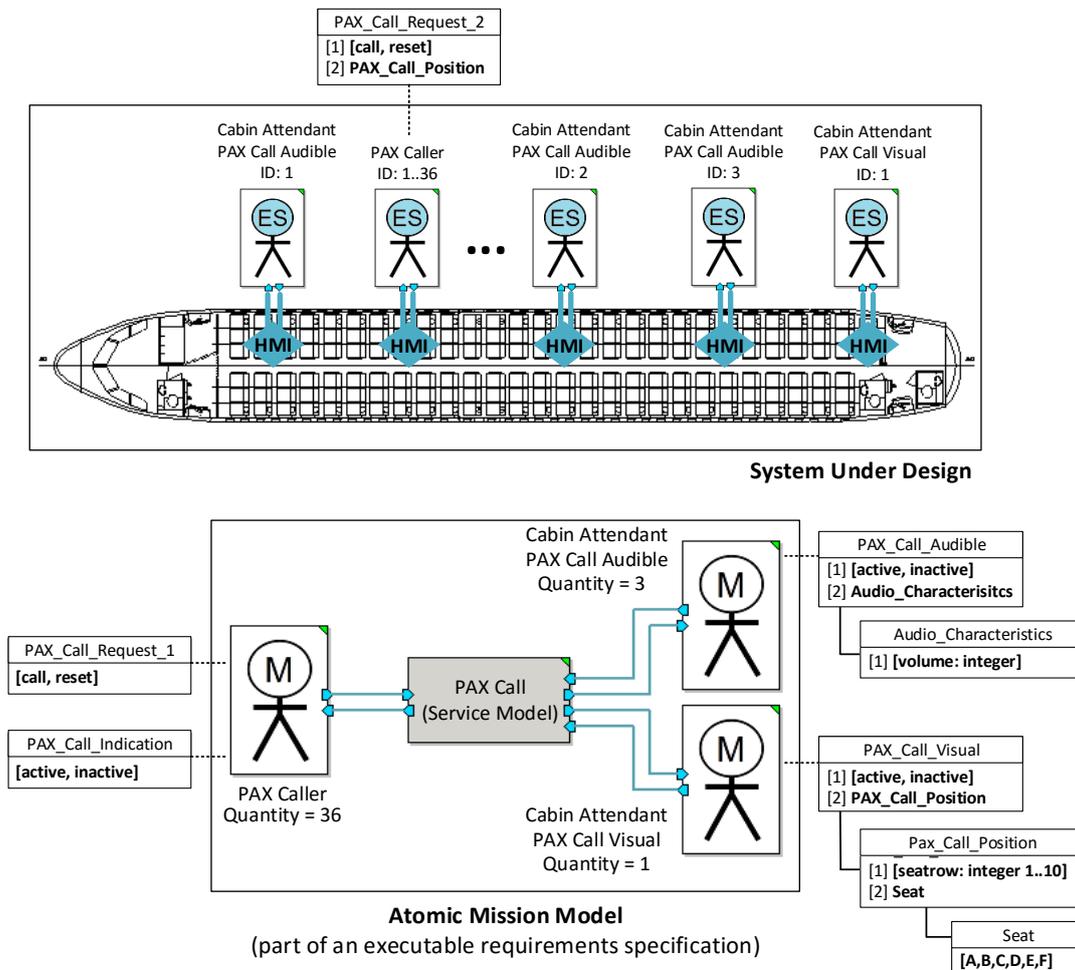


Figure 5.9 – Atomic mission model example for use case *PAX Call* with central use case node, different AMIS actors (bottom) and ES actors (top), intended data flow (cyan lines), data model (actor adjacent) and interaction points of type HMI (cyan rhombuses)

The data model for actor “*Cabin Attendant PAX Call Audible*” is composed of an enumeration to indicate the status of audible information (active or inactive) as well as of an auxiliary composite data structure called *Audio_Characteristics* that is used to determine a specific characteristic of the audible information provided during a PAX Call (audio volume). As part of this characteristic, a bounded interval for possible values is determined in order to indicate a non-functional requirement with specific range. For instance, data structure member “*volume*” is defined to be an integer number between 60 to 100 that represents a decibel range requirement for audible indication performance. Actor “*Cabin Attendant PAX Call Visual*” also uses an enumeration to determine possible states during the use case (active or inactive)

and an auxiliary composite data structure called *PAX_Call_Position*. This composite structure is used as part of the intended use case in order to provide cabin attendants with more specific information about the source of a PAX Call. The structure contains a seat row indication with range 1 to 10 as well as an enumeration of characters that refer to specific seat positions within a single aisle aircraft cabin with three related seats on each side (A to F). For AMIS actor “*PAX Caller*”, two different data structures have been determined. *PAX_Call_Request_1* is required to activate and deactivate a PAX Call (call and reset). On the other hand, *PAX_Call_Indication* is used to provide passengers with feedback on the status of a PAX Call (active, inactive). Each AMIS actor is also provided with a specific value for the parameter quantity. In consequence, the same number of ES actors with associated IDs is created for each AMIS actor type. In order to avoid a huge number of single ES actors, it is possible to determine aggregated ES actors with ID range, e.g. ES actor “*PAX Caller*” shown within Figure 5.9.

In parallel, complementary ES actors are determined with associated name and ID parameters (see top of Figure 5.9). For all cabin attendant actors, the data model of the corresponding AMIS actors is inherited. In the case of the ES actor of type “*PAX Caller*”, the data model needs to be extended, i.e. data structure *PAX_Call_Request_1* is complemented by a second data structure of type *PAX_Call_Position*, which is the same auxiliary composite data structure that is used for actor “*Cabin Attendant PAX Call Visual*”. This is necessary because of the intention of the use case to indicate the specific location of a PAX Call to cabin attendants. A fact that is supported by the already defined data model for actor “*Cabin Attendant PAX Call Visual*”. As a result, the ES actor of type “*PAX Caller*” uses an extended data model of the corresponding AMIS actor called *PAX_Call_Request_2*. This data model extension is fed back to the associated AMIS actor in order to ensure a consistent data model (not shown in Figure 5.9).

Figure 5.10 depicts the top-level view of the completed AMIS model of use case PAX Call together with already determined parameter values for each actor.

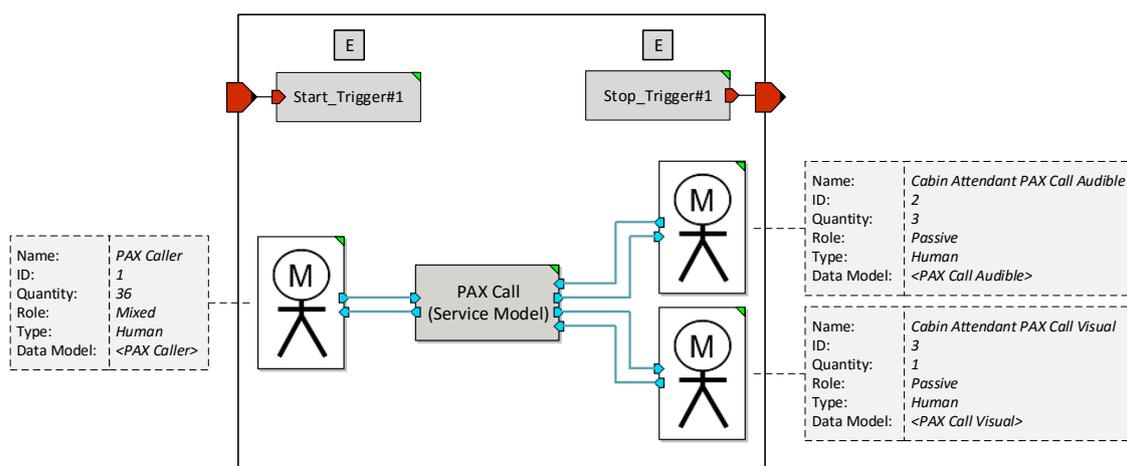


Figure 5.10 – Atomic mission model with parameters (boxes with dashed lines) for use case *PAX Call* of a basic cabin management system

Our second example considers a use case called “provide lavatory smoke indication” or, in short, “*LAV Smoke*”. Figure 5.11 depicts the top-level view of the completed

AMIS model of use case LAV Smoke. In this use case, a smoke detection warning shall be issued to all members of the cabin crew in case a fire is detected within the trash bin of a lavatory. In this example, two actors are determined. One human actor called “*Cabin Attendant LAV Smoke*” and one non-human actor called “*LAV Smoke Source*”.

“*LAV Smoke Source*” represents an abstract environment state rather than an actor entity with specific goals, e.g. human actors or system components. Moreover, actor “*LAV Smoke Source*” acts as initiator of the overall use case (precondition), since smoke needs to be present in order to be detected and indicated. This actor is different from all actors that were used within example one since it represents an environment interaction entity on the boundary of the SuD that is associated with an interaction point in the form of a sensor instead of an HMI. So instead of providing a second actor of type sensor, an actor of type environment was chosen that is associated with a system interaction point that is likely to be some sort of sensor. This is because actors of atomic mission models and overall system models operate with interaction points that are on the boundary of the system under design. Thus, they either use services or provide conditions for services in order to be able to validate specific system designs. The data model for LAV Smoke is shown with more detail as part of the following paragraph on customization in Figure 5.13.

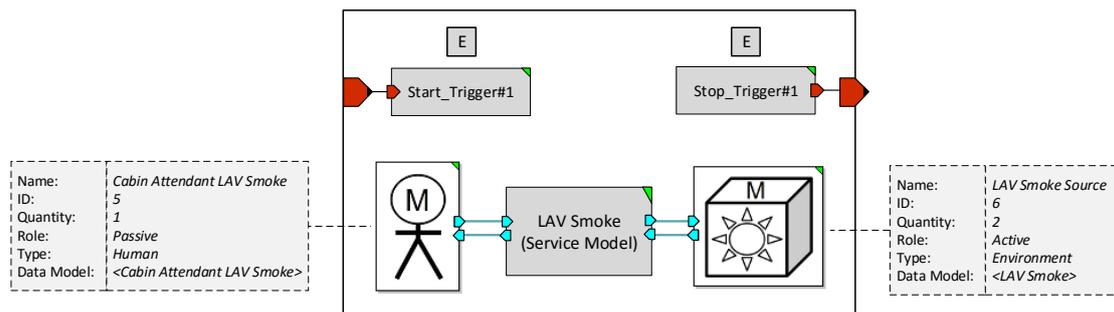


Figure 5.11 – Atomic mission model with parameters (boxes with dashed lines) for use case *LAV Smoke* of a basic cabin management system

Similar to example one, necessary AMIS and ES actors are determined. Again, visual indications shall be indicated at different public areas within the cabin in case smoke has been detected. Accordingly, a data model is developed that includes enumerations to indicate the status of smoke detection as well as auxiliary composite data structures that can be used to pinpoint the exact location of a smoke alarm. In the case of actor “*LAV Smoke Source*”, an abstract data model can be created that includes an enumeration to indicate an environment state input, e.g. *smoke* or *no smoke*. Moreover, the data structure can be extended by adding specific quantifiable data parameters, e.g. *temperature*. This is dependent on the overall intention of the use case as determined by the conceptual design.

Customization Model Development

In the case of CMS for large aircraft, it is important to consider and evaluate design alternatives and system customization aspects early during design (cf. chapter 1.1). Other than CMS customization, the process of CMS configuration is performed during product planning and manufacturing for specific aircraft for each customer (cf.

chapter 1.1). As part of this process, a subset of all possible services is compiled together with specific architecture configurations that determine number and characteristics of system equipment, e.g. due to specific numbers and arrangements of seats and passenger service units. In preparation for system configuration, minimum and maximum system configurations for services with associated actors are determined and validated as part of overall mission model development. By creating minimum and maximum versions for atomic mission models with according number of actors and adapted service models, it is possible to determine and validate bounded ranges for possible system characteristics. These ranges provide the foundation for system configuration later during system development and manufacturing.

As elaborated before, different types of customization parameters exist (cf. chapter 4.3.2.2). The differences between the three major customization parameter categories are illustrated in the next few sections by continuing the two atomic mission model examples described in the last paragraph. In addition, a third use case example for the top-level use case “*provide status information for passengers*” is used. It describes an AMIS model for automated operation of visual fasten seatbelt (FSB) indication during aircraft operation.

As part of the atomic mission model for use case “*PAX Call*” that was described earlier, cabin attendants are notified about active passenger calls via visual indications. In this case, a fixed design parameter is assumed that has already been determined beginning with concept design, preliminary architecture design and atomic mission model development. Active PAX calls are indicated with an indication light (active, inactive) at the respective seat row. This decision also decreases the design space for human machine interface (HMI) solutions, since a specific way of event indication has already been determined implicitly. As part of the same example, another fixed design parameter is already part of the data model for actor “*Cabin Attendant PAX Call Audible*”. This parameter is an audio characteristic labeled “volume” (cf. Figure 5.9).

When extending the above example to include variable design parameters for customization of architectural characteristics, it may be assumed that indication lights for use case “*PAX Call*” are colored. Hence, an enumerated customization parameter is determined that contains a limited set of available colors, e.g. {blue, green, yellow} with “blue” being the default value. Another parameter describes different indication modes, e.g. for steady or blinking indication lights with additional parameters for blink duration and frequency. In this example, human actors expect certain visual qualities, e.g. specific indication light colors with specific indication mode, that can be observed at an HMI interaction point that provides a specific functional output (PAX Call indication). Thus, the data model of actor “*Cabin Attendant PAX Call Visual*” is extended by quality-related customization data structures for indication light colors and different indication light modes as shown in Figure 5.12.

In a second example, the atomic mission model for use case “*LAV Smoke*” shall be extended with an optional customization parameter for functional characteristics. In this case, an optional selectable audio chime is defined that is to be played in the case of lavatory smoke detection in addition to visual indications. For this purpose, a customization parameter called “*Cabin Chime Activation LAV Smoke*” is developed with enumerated type Boolean {true, false}. Moreover, an additional non-functional customization parameter is determined for chime characteristics, e.g. in order to

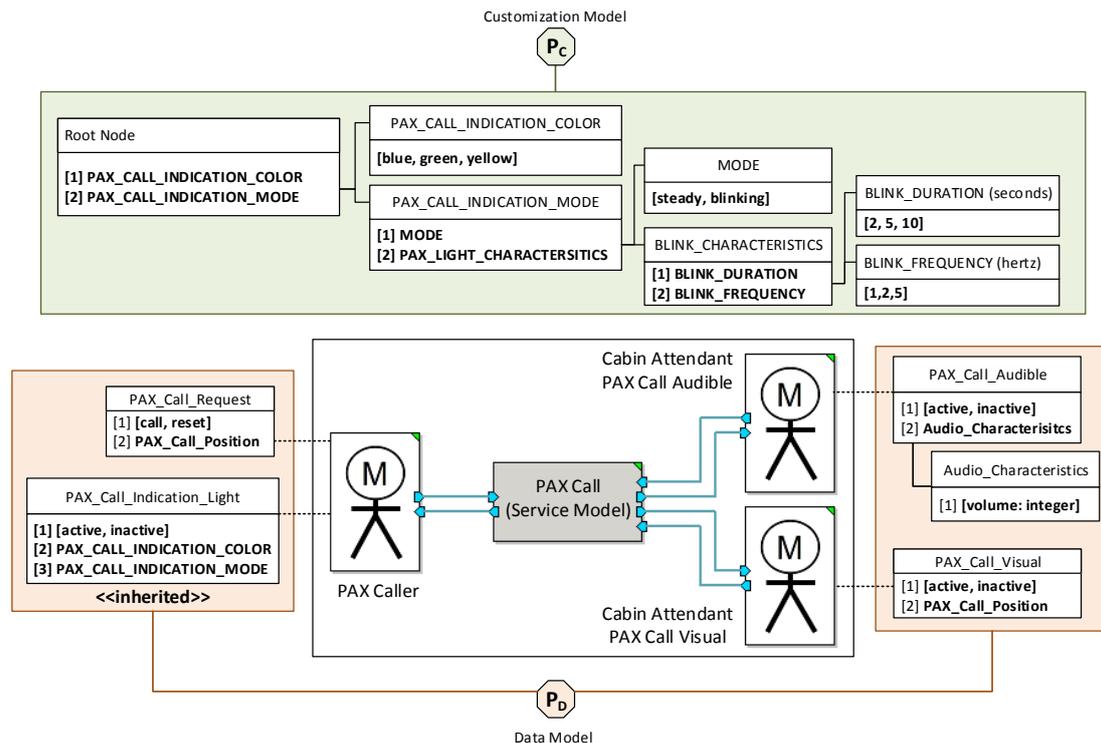


Figure 5.12 – Atomic mission model example for use case *PAX Call* with customization model (top) and updated data model (bottom) with inherited data structures [2] and [3] as part of *PAX_Call_Indication_Light*

provide a set of different sounds, durations and volumes. As a result, an additional functional characteristic is expected by actors in case a certain configuration has been chosen (chime indication active or inactive). In this case, additional quality characteristics are linked to the optional functional customization parameter (chime attributes). As depicted in Figure 5.13, two different options were considered.

Option one is to extend the data model of actor “*Cabin Attendant LAV Smoke*” with a new data structure “*LAV_Smoke_Indication_Chime*” with members [1] = {*chime_in_progress*, *chime_inactive*} and [2] = {*LAV_SMOKE_CHIME_CHARACTERISTICS*}. The second member is directly inherited from the corresponding quality customization parameter for chime characteristics (*chime type*, *volume*). The first member is derived from the functional customization possibility to enable chime indications as part of the atomic mission model. Thus, actors need to be able to observe if a chime is being played or not at a specific point of time.

Option two is to create a new pair of actors (AMIS and ES actors) that represent an actor linked to an audio HMI interaction point. This actor is called “*Cabin Attendant LAV Smoke Audible*”. The final decision for a specific option is determined based on the design concept and preliminary system architecture.

The usage of conditions and constraint-based customization parameters shall be illustrated by using another example from the context of CMS. Consider the example of fasten seatbelt (FSB) indication for aircraft passengers and cabin crew. Normally, FSB indication is switched on and off by members of the cockpit in order to advise people within the cabin to remain seated with seatbelts fastened, e.g. during take-off and landing or during turbulences. In addition, cockpit personnel may activate an automatic FSB indication mode. In this case, the CMS dynamically determines

whether FSB indication is to be switched on or off during flight, based on the evaluation of different indicators or linked events, e.g. from external systems. As part of a customization model, the combination of events that lead to active or inactive FSB indications may differ for each customer because of different flight operation models or because of different types of aircraft that use different event indicators. In other cases, more than one event is combined in order to gain higher confidence when determining the current flight phase during operation.

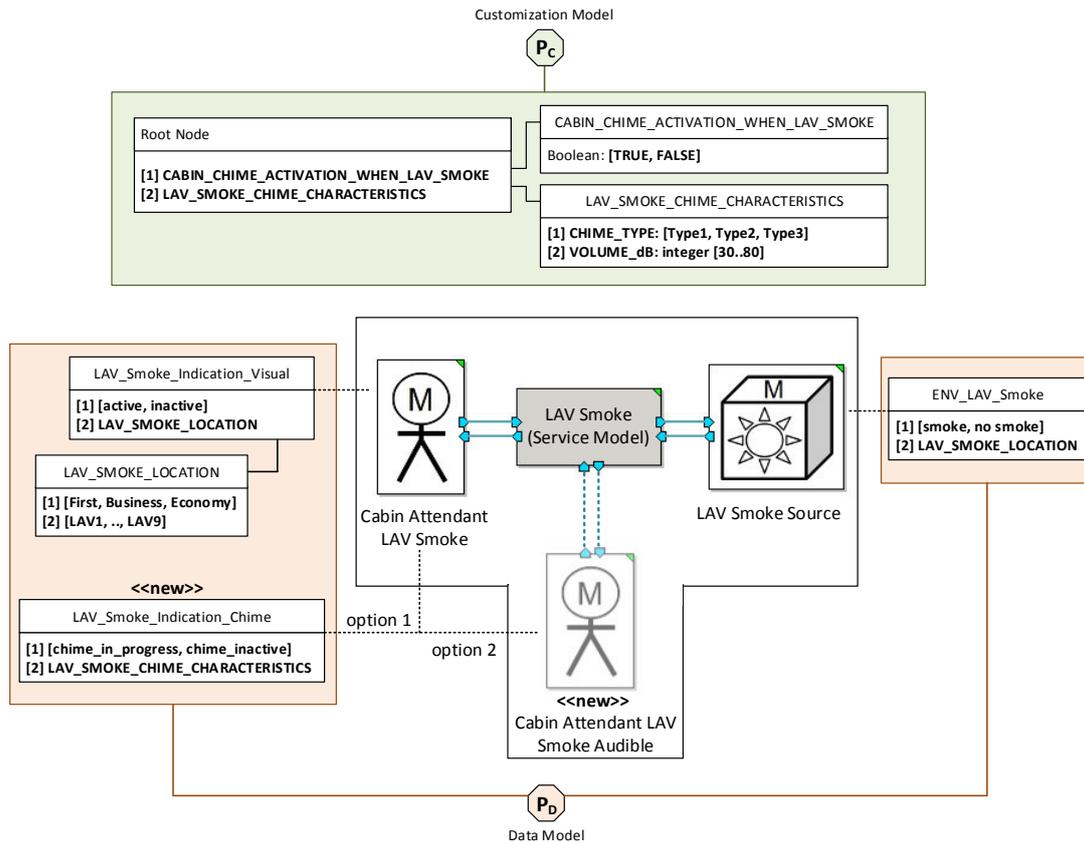


Figure 5.13 – Atomic mission model example for use case *LAV Smoke* with customization model (top) and updated data model (bottom) with new data model *LAV_Smoke_Indication_Chime* as part of actor *Cabin Attendant LAV Smoke* (option 1) or part of an additionally introduced actor *Cabin Attendant LAV Smoke Audible* (option 2)

In this customization example, an atomic mission model is considered that describes an automatic FSB indication service with a conditional customization parameter that allows to choose different event conditions that will activate FSB indication within the aircraft cabin. As part of this atomic mission model, cabin crew as well as passengers will be visually informed about the current status of FSB indication. Events taken into consideration include the statuses of landing gears (actor *LG Sys*), engines (actor *ENG Sys*) and a safety-relevant event that is the total loss of cabin pressure (actor *CPCS*). As a result, customization parameter *FSB_ON_Auto_Conditions* is used with three different logical customization expressions *LCE1...LCE3*:

$FSB_ON_Auto_Conditions = \{(1, LCE1), (2, LCE2), (3, LCE3)\}$, where:

$LCE1 := LGSys:Landing_Gear_Down_and_Locked$
 $LCE2 := LGSys:Landing_Gear_Down_and_Locked \ \&\& \ ENGsys:Engines_ON$
 $LCE3 := (LGSys:Landing_Gear_Down_and_Locked \ \&\& \ ENGsys:Engines_ON) \ || \ (CPCS:Loss_of_Cabin_Pressure)$

If parameter member $LCE1$ is chosen for customization, FSB indication shall be switched on if event *Landing_Gear_Down_and_Locked* of actor *LGSys* (landing gear system) has been set to “TRUE”. In case parameter member $LCE2$ is selected, FSB indication shall only be switched on if, in addition to fulfillment of $LCE1$, event *Engines_ON* of actor *ENGsys* (engine system) has been set to “TRUE”. If member $LCE3$ is chosen as customization parameter setting, FSB indication shall be activated in case $LCE2$ is fulfilled and/or event *Loss_of_Cabin_Pressure* of actor *CPCS* (cabin pressure control system) has been set to “TRUE”.

Figure 5.14 depicts an atomic mission model for the example of an automatic FSB indication service together with customization and data models. The customization parameters shown represent functional event conditions and function-relevant actor assignments.

In addition to the actors and events described before for condition-based customization parameter $FSB_ON_Auto_Conditions$, two different actors can be chosen to activate the use case of automatic fasten seatbelt indication (AUTO FSB Indication). Both actors, *Cockpit FSB 1* and *Cockpit FSB 2*, share the same data model. Via data structure *FSB_Auto_Indication_Control*, each of the two actors is able to start or end the use case *AUTO FSB Indication*. If *FSB_Auto_Indication_Control* is set “on”, the use case is triggered and inputs of actors *LGSys*, *ENGsys* and *CPCS* will be evaluated in accordance with the logical expression provided by customization parameter $FSB_ON_Auto_Conditions$. As a result, FSB indication within the cabin will be switched on or off. The resulting behavior of visual HMIs be observed by cabin crew and passenger actors. Setting *FSB_Auto_Indication_Control* “off” will end the overall use case. Customization parameter $FSB_Control_Assignment$ is used with three different logical actor assignment expressions $LAAE1...LAAE3$:

$FSB_ON_Auto_Conditions = \{(1, LAAE1), (2, LAAE2), (3, LAAE3)\}$, where:

$LAAE1 := Cockpit_FSB_1$
 $LAAE2 := Cockpit_FSB_2$
 $LAAE3 := Cockpit_FSB_1 \ || \ Cockpit_FSB_2$

If $LAAE1$ has been chosen for customization, only actor *Cockpit_FSB_1* will be able to trigger or end use case *AUTO FSB Indication*. As part of a second customization possibility, $LAAE2$ is determined in order to enable actor *Cockpit_FSB_2* to trigger or end the use case. When choosing $LAAE3$, both actors are able to trigger or end the use case. As a result of this last parameter option, developers of the service model for *AUTO FSB Indication* need to determine specific functional properties with respect to possible combinations of data structure instances of each of the two actors. For instance, to handle a sequence of events beginning with actor *Cockpit_FSB_1* where *FSB_Auto_Indication_Control* = “on” followed by an event of actor *Cockpit_FSB_2* where *FSB_Auto_Indication_Control* = “off”.

As a consequence of customization model development for automated FSB indication, models of later development steps, e.g. service models or executable overall system specification, need to provide means to support all three different event conditions. Since FSB indication needs to be provided to aircraft passengers continuously from boarding to disembarkment, this atomic mission model can also be modeled in the form of a continuous atomic mission model. In this case, the respective atomic mission model is represented as a detached node within the associated mission model.

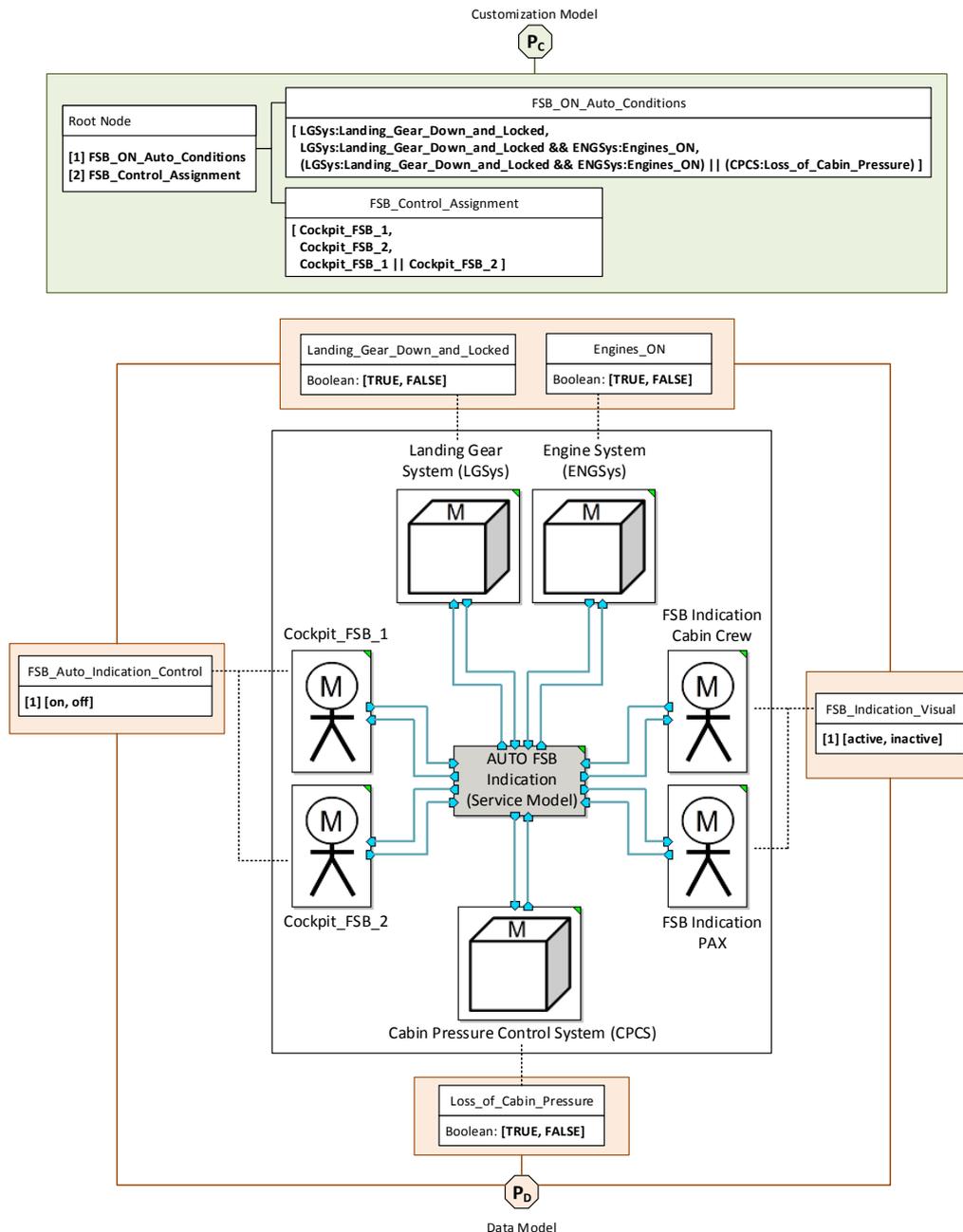


Figure 5.14 – Atomic mission model for an automated FSB indication service, including customization parameters for function-relevant actor assignments and event conditions

Figure 5.15 shows an updated version for use case example *LAV Smoke* with one customization actor (top) for customization model exchange. A full overview of the developed data and customization model is provided in appendix B.3.

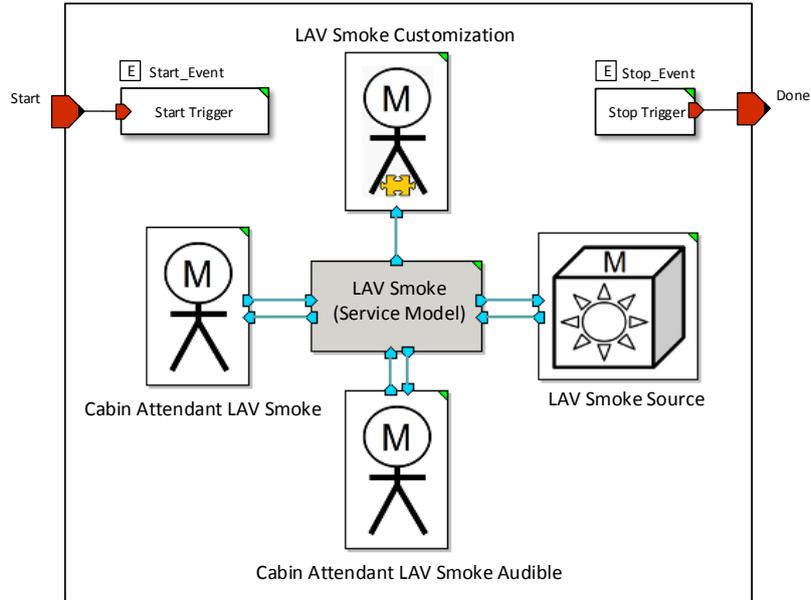


Figure 5.15 – Updated atomic mission model example for use case *LAV Smoke* including a dedicated customization actor (top)

5.4.3 CMS Quality Objective Model

Global non-functional or quality requirements are determined at mission level. In the example mission model for a cabin management system (CMS) depicted in Figure 5.8 of chapter 5.4.1, one continuous quality objective models (QOM) has been determined for non-functional requirement “overall power” together with two static QOMs for non-functional requirements “overall system weight” and “overall system cost”. Accordingly, three quality objectives (QO) have been determined. Each quality objective is described in the form of bounded ranges by a 4-tuple (n, lb, val, ub) , where n is the name of the objective, lb is the lower bound value, v is the intended mean or target value and ub is the upper bound value.

Static quality objective “overall system weight” has been determined in kilogram with objective $o_1 = (weight, 0, 400, 600)$. The target or expected value for weight is specified as 400 kg, but any value between 0 kg and 600 kg will be regarded as acceptable. The second static quality requirement for overall system cost was determined in US dollar with objective $o_2 = (cost, 0, 20000, 80000)$ whereas the dynamical objective for overall system electrical power consumption in watts was determined with objective $o_3 = (power, 100, 2000, 7000)$. In this case, the objective has a lower bound value greater than zero because the associated CMS will always require a minimum amount of electrical power when in service or in standby mode. This is due to a set of essential CMS functions that are required for reasons of safety, e.g. the possibility to perform a public address service at any time of aircraft operation. Together with the creation of OQMs, associated non-functional (NF) observer modules are configured for later use, i.e. during executable overall specification development or automated validation (cf. chapter 5.5). Figure 5.16 depicts a section of a passenger service unit (PSU) element that is part of an associated executable system specification for a cabin management system which was created during later

development stages. One sub-element is depicted in Figure 5.16 that, in this case, represents a human machine interface (HMI). This element includes no smoking indications (left) and fasten seatbelt indications (right) and is also regarded a system interaction point that is associated with different atomic mission models. As a result, this element is coupled with executable specification actors (ES actors). Moreover, different PSU input and output ports (left-hand side) and transitions (solid lines) are shown.

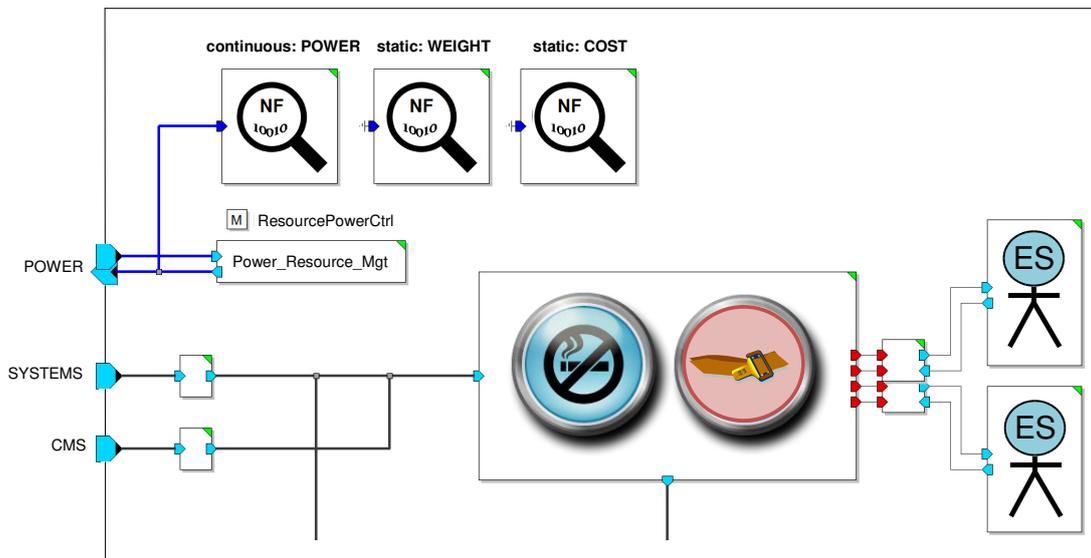


Figure 5.16 – Section of a passenger service unit element that is part of an executable system specification for a cabin management system in MLDesigner with three different non-functional observer modules (top left) and different ES actors

A non-functional observer module (first NF observer module at the top) is connected to a transition that relays information on the power consumption of each PSU element and its sub-components during simulation (power interface). This non-functional observer module, in combination with other instances of the same type, continuously monitors overall power consumption during simulation and is used for validation by the continuous quality objective model shown in Figure 5.8 for non-functional requirement “overall power”.

The other two non-functional observer modules depicted in Figure 5.16 (second and third NF observer module at the top) are not connected to any component or transition via their input port (terminated port). Instead, they are directly coupled with overall PSU element parameters for “weight” and “cost”. They are connected to respective static quality objective models depicted in Figure 5.8 for non-functional requirements “overall system weight” and “overall system cost”. Note, that parameter values of elements can also be changed dynamically during simulation as a result of functions that are performed as part of an element. In this case, direct parameter to non-functional observer linking can be used to validate continuous quality objective models.

During automated CMS specification validation (cf. chapter 5.5), non-functional design properties “overall system weight” and “overall system cost” are validated once, based on the data received from non-functional observer probes of the associated

executable overall system specification. In contrast, non-functional design property “overall power” is validated continuously throughout overall mission execution.

5.4.4 CMS Environment Configuration Model Development

Since a cabin management system (CMS) for large aircraft is an integrated system that has interfaces to many aircraft subsystems, a wide variety of system and system environment configuration parameters exist.

When determining system or system environment configurations, it is important to separate global configuration parameter that will possibly affect a number of different systems and use cases from local parameters that do only affect a specific part of a system under design. Reconsidering the example of a smoke detection service for aircraft lavatories, it can be observed that actor “*LAV Smoke Source*” represents an abstract local environment state that, in this case, is unrelated to any other use case and thus has no side effects on other atomic mission models.

However, if the event of smoke detection with related use case would trigger other events that would lead to an impact on the overall mission, e.g. the initiation of different emergency procedures, the same environment state would become a global configuration object at mission level. In another example for the development and validation of a cabin video monitoring system for large aircraft, a use case for cabin observation may require to adjust video quality in response to changes in ambient lighting. Again, environment configurations will only affect one specific use case. As a result, actors for local environment states can be used to alter cabin environment parameters instead of environment changes at overall mission level.

In the example mission model depicted in Figure 5.8 of chapter 5.4.1, three different environment configuration models (ENV) are depicted. The first two ENV modules called “Aircraft on Ground” and “Electrical Power ON (preflight)” are used to change to single environment states. This is done by configuring the parameters *Reference System or Environment Name* and *Parameter to Change* of the associated system environment configuration modules. ENV module “Aircraft on Ground” is set to (*aircraft_environment, on_ground*) while ENV module “Electrical Power ON (preflight)” is set to (*power_plant, A/C_power_on_preflight*). When executed, the CMS environment is changed so that the overall aircraft is considered to be on ground with all power plants active (engines on). The third ENV module depicted in Figure 5.8 is a custom system environment configuration module called “Complete Flight Phase Cycle”. Figure 5.17 illustrates the internal structure of this custom system environment configuration module for CMS mission models. It is composed of a finite state machine (FSM) for system environment configuration and an environment configuration change module of type C3 (cf. chapter 4.3.4).

The statechart module depicted in Figure 5.17 is used to cycle through different flight phases of a commercial aircraft in order to determine associated system environment configuration parameter changes. When triggered, the flight phase FSM changes its state as specified and produces a data structure instance of the system environment configuration model E_C . This data structure includes information for different system environment model entities. Thus, it is possible to change a more complex set of aircraft parameters within one event, e.g. current height and speed, as well as internal states of different other subsystems, e.g. the statuses of landing

gear and power provisioning systems. The data produced is used by the predefined C3 module in Figure 5.17 to change the system configuration of an associated executable overall system specification.

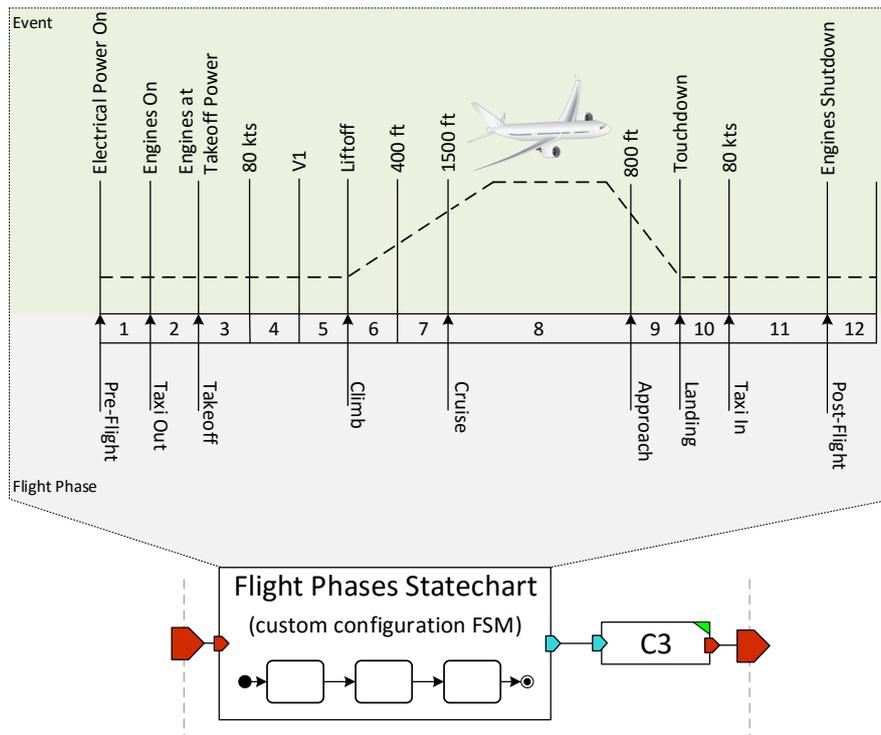


Figure 5.17 – Custom system environment configuration module for aircraft flight phases

5.4.5 CMS Service Model Development

After atomic mission (AMIS) model development has been completed, service models are developed for each AMIS model. Each service model describes an executable flow of different service scenarios, usually divided between one main scenario and a finite set of optional or alternative scenarios. It is also possible to determine different scenarios for different customization parameter configurations.

Consider the example of the service called *“provide lavatory smoke indication”*, in short *“LAV Smoke”*, that has been discussed within the last sections. In Figure 5.18, a sequential service model for the atomic mission model of *“LAV Smoke”* is depicted. On the upper side of the model, four Actor Data Exchange (ADE) modules are shown that are connected to actors of the associated upper level AMIS model shown in Figure 5.15 of chapter 5.4.2. The turquoise ADE module on the right-hand side is used to exchange customization data via the associated customization actor. The first scenario model shown in Figure 5.18 determines a service scenario that uses a customization model with no audible indication chime. All following scenario models use customization models with audible indication chime and different chime types. In this example, no additional quality objective model is used.

As elaborated before, the risk of scenario model explosion increases with the number of potential customization parameters. Consider the example of an atomic mission model for the service *“call cabin attendant for assistance”* (PAX Call) depicted in Figure 5.12 of chapter 5.4.2. In this example, two high level customization parameters exist to configure PAX Call indication characteristics. These two parameters

branch out into a set of different configurable sub-parameters, leading to a total of 54 possible customization parameter value combinations with a subset of 30 useful combinations. Thus, designing a scenario model for each of the useful combinations would lead to 30 scenario models. Although this example will cause no problem in terms of clarity and model execution efficiency, service models with thousands of scenario models may prove inadequate.

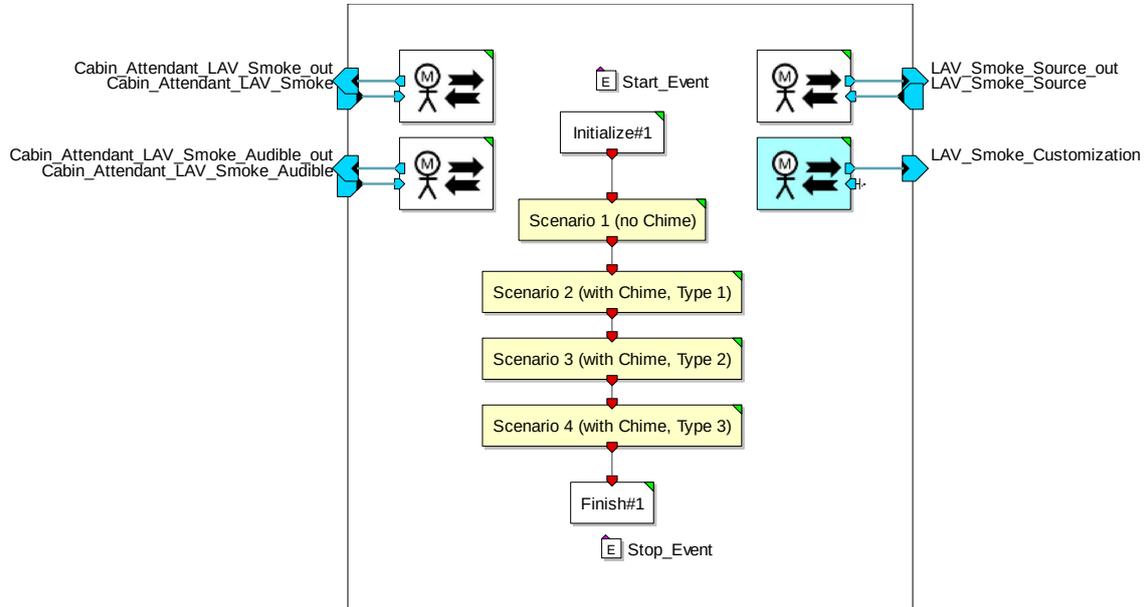


Figure 5.18 – Service model example for atomic mission model *LAV Smoke*

After main and alternative scenarios models have been determined as part of an associated service model, each of the scenarios needs to be modeled in detail. Most important part of this process is the development of associated scenario flowchart (SFC) modules. These modules represent driver and evaluation models for the system under design, i.e. the overall system specification (EOSS) that is executed together with the overall mission model during automated validation. In the next few sections, a step by step SFC example is provided for use case “*PAX Call*”. Figure 5.19 depicts the overall SFC model that is part of a service model for “*PAX Call*” in MLDesigner. It is depicted in vertical arrangement for reasons of clarity.

Three parameters are provided for the example depicted in Figure 5.19. Parameter *Time_Out* determines a configurable value for an overall timeout in order to avoid deadlocks, e.g. in case no system responses occur. *Wait_Objective* determines a configurable range that describes a waiting period that is used by one of the expected system response states. *Chime_Volume* determines a quality objective for chimes that are triggered with every *PAX Call* activation.

In general, transitions are labeled with an event that will trigger a state change as well as optional actions. Other transitions have corresponding alphabetical labels (*A-E*) and are described in the following paragraphs. Transitions with label *e* are synchronous transitions.

All three mayor states *INACTIVE*, *ACTIVE* and *FINISHED* are depicted by grey ovals while other sub-states use white ovals. For reasons of clarity, actor stimuli sequence processes are colored in blue while expected system response processes are

colored in green. One control flow input port t and one control flow output port d are provided.

Derived from the superordinate AMIS model, data ports for four different actors with corresponding typecast are provided. These include three input ports for actors *PAX Caller*, *Cabin Attendant PAX Call Audible* and *Cabin Attendant PAX Call Visual* as well as two output ports for actors *PAX Call Customization* and *PAX Caller*. Above each data port, a number or range determines the multiplicity of associated AMIS model actors.

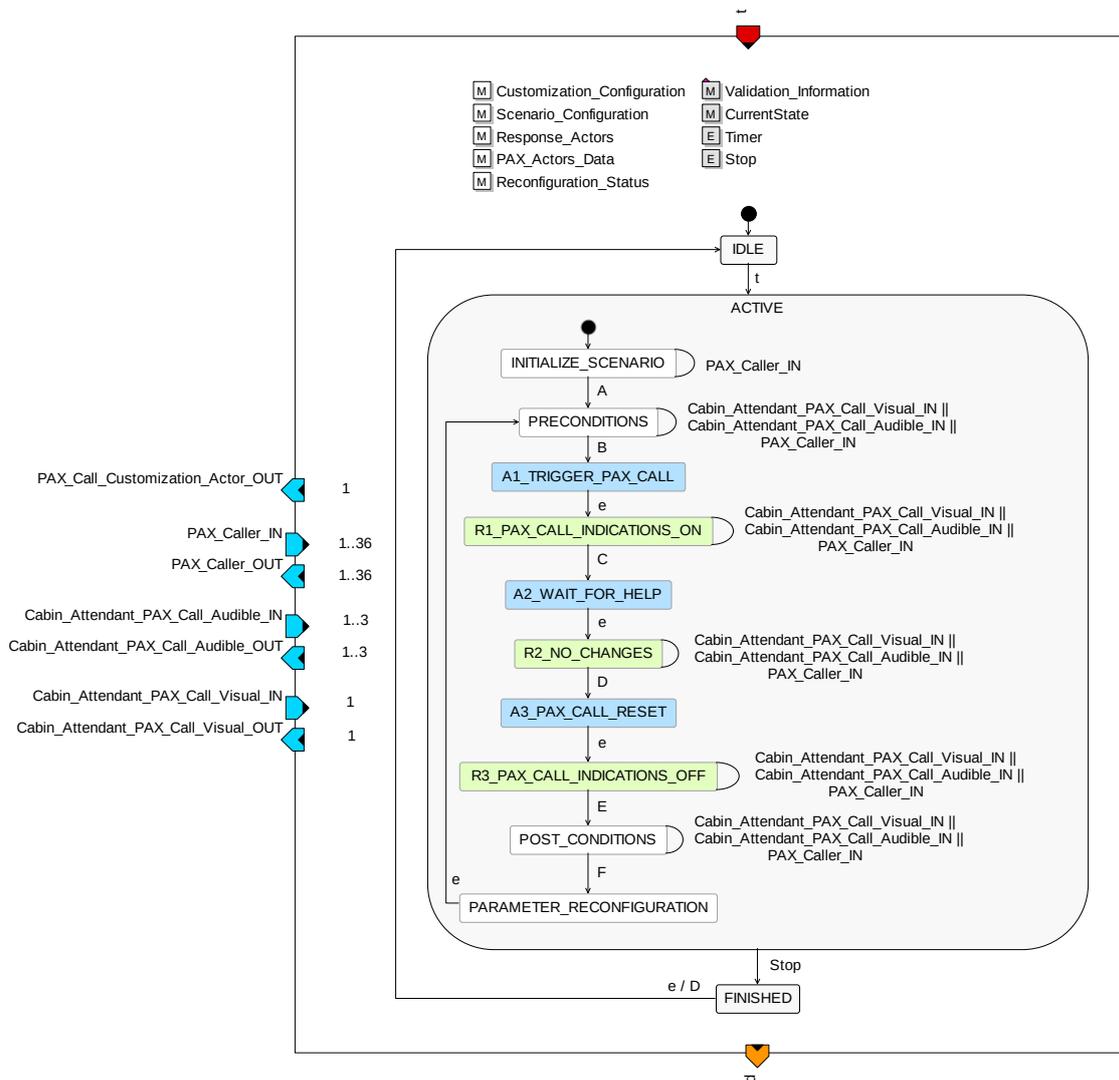


Figure 5.19 – Scenario flowchart example (vertical arrangement) for atomic mission model *PAX Call* in MLDesigner

In addition, six internal memories IM , one external memory EM and two events E are used in the example depicted in Figure 5.19. Components shaded in gray are not initialized and are required for internal operation of the SFC. All nine components shall be described briefly:

- *IM1 Customization_Configuration*: Contains current customization parameter configurations.

- *IM2 Scenario_Configuration*: Contains current scenario parameter configurations. In this case, the current PAX Caller actor that is used for actor stimuli sequences.
- *IM3 Response_Actors*: Used by expected system response activities in order to save and track EOSS responses.
- *IM4 PAX_Actors_Data*: Used by expected system response activities in order to evaluate EOSS responses regarding the specific PAX call position of an associated PAX call actor.
- *IM5 Reconfiguration_Status*: Used by states *PostCondition* and *Parameter_Reconfiguration* to determine if all necessary configurations of customization parameters and scenario parameters have been performed.
- *IM6 CurrentState*: Internal memory used by MLDesigner.
- *EM Validation_Information*: Used by all states in order to provide validation information for a global validation report.
- *E Timer*: Used to trigger transitions after a specific amount of time.
- *E Stop*: Used to terminate the execution of the overall SFC.

During scenario flowchart execution, the following set of states is executed top-down:

1. Inactive: *IDLE*
2. Active: *ACTIVE*
 - 2.1. Initialization: *INITIALIZE_SCENARIO*
 - 2.2. Preconditions: *PRECONDITIONS*
 - 2.3. Actor Stimuli Sequence: *A1_TRIGGER_PAX_CALL*
 - 2.4. Expected SuD Responses: *R1_PAX_CALL_INDICATIONS_ON*
 - 2.5. Actor Stimuli Sequence: *A2_WAIT_FOR_HELP*
 - 2.6. Expected SuD Responses: *R2_NO_CHANGES*
 - 2.7. Actor Stimuli Sequence: *A3_PAX_CALL_RESET*
 - 2.8. Expected SuD Responses: *R3_PAX_CALL_INDICATIONS_OFF*
 - 2.9. Post-Conditions: *POST_CONDITIONS*
 - 2.10. Parameter Reconfiguration: *PARAMETER_RECONFIGURATION*
3. Finished: *FINISHED*

In the following paragraph, the characteristics of each state are described in detail by using pseudo code. A full MLDesigner specification of this scenario flowchart is provided in appendix B.2. During overall system simulation, i.e. during automated validation, the scenario is idle and waits for a control token trigger in state 1).

Algorithm 1): State IDLE

```

1 if Input: t
2 then
  | Output: Transition t to ACTIVE
3 else
4 | WAIT

```

Algorithm 2): State ACTIVE

```

1 while in state ACTIVE do
  | // Stop Event active by timeout or validation failed
2   if E_Stop == true.failed || E_Stop == true.timeout then
3     | M_Validation_Information ← validation failed information
4   else
5     | M_Validation_Information ← validation successful information
  | Output: Transition Stop to FINISHED

```

When in state 1) and triggered successfully, a transition to superordinate state 2) is activated. When active, state 2) observes overall scenario timeout, validation failed and validation successful events. Upon activation of any such event, the ending of the overall SFC is prepared by passing over to state 3).

When in state 2), sub-state 2.1) is used to initialize the overall scenario. Therefore, memories are initialized and a timeout event is set to avoid deadlocks. At first, a default customization model setting is determined and stored within memory *Customization_Configuration*. Subsequently, the default customization setting is sent to an associated customization actor in order to provide a common base for mission and EOSS models.

Algorithm 2.1): State INITIALIZE_SCENARIO

```

  // initialize global timeout Event
1 E_Stop ← timer Time_Out
  // initialize Customization Model configuration
2 M_Customization_Configuration ← default customization configuration
  Output: A_PAX_Call_Customization ← M_Customization_Configuration
  // initialize other memories
3 M_Reconfiguration_Status ← nil
  // memories are used to store data for actors, stimuli and responses
4 M_Scenario_Configuration ← current actor PAX_Caller1
5 M_Response_Actors ← initialize vector of length 40 (number of possible response actors)
6 M_PAX_Actors_Data ← initialize vector of length 36 (number of possible PAX Caller actors)
7 for i = 1 to 36 do
  | Output: PAX_Caller ← request current position data of PAX_Calleri
8 while Input: PAX_Caller
9 do
10 | index ← PAX_Caller.getID
11 | if M_PAX_Actors_Dataindex == empty then
12 | | M_PAX_Actors_Dataindex ← PAX_Caller
13 | else
14 | | M_Validation_Information ← validation information E_Stop ← true.failed
15 | if M_PAX_Actors_Data filled successfully then
16 | | M_Scenario_Configuration ← update data structure PAX_Call_Position from M_PAX_Actors_Data
  | Output: Transition A to PRECONDITIONS

```

A scenario configuration memory is used to determine a specific actor of type *PAX Caller*. In this case, actor *PAX_Caller*₁. This specific actor is used during stimuli sequences and for evaluating SuD responses. Moreover, this memory is required for

scenario reconfiguration later during state 2.10). As described in chapter 4.3.2.1, specific EOSS model actors are addressed via name and unique ID. In order to save the specific locations of each single actor of type *PAX Caller*, current actor configurations (*Pax_Call_Position*) are requested from the associated EOSS model and stored accordingly. If successful, the next state in line is activated via transition A. Otherwise, a timeout event is triggered immediately.

In state 2.2), preconditions for the PAX Call scenario are determined and evaluated. In general, a check is performed in order to ensure, that no PAX Call is active at this stage of scenario execution. In order to do so, current states of all actors *Cabin_Attendant_PAX_Call_Visual*, *Cabin_Attendant_PAX_Call_Audible* and *PAX_Caller* are requested from the associated EOSS model and stored. Each EOSS actor shall provide exactly one status response with indication status “inactive”. If all responses have been received successfully, transition B is executed.

Algorithm 2.2): State PRECONDITIONS

```

// reset timer Event
1 E_Stop ← timer Time_Out
  // check that no PAX Call is currently active
  Output: Cabin_Attendant_PAX_Call_Visual ← request current status of
          Cabin_Attendant_PAX_Call_Visual1
2 for i = 1 to 3 do
  | Output: Cabin_Attendant_PAX_Call_Audible ← request current status of
  | Cabin_Attendant_PAX_Call_Audiblei
3 for i = 1 to 36 do
  | Output: PAX_Caller ← request current status of PAX_Calleri
4 while Input: Cabin_Attendant_PAX_Call_Visual || Cabin_Attendant_PAX_Call_Audible || PAX_Caller
5 do
6   input ← Cabin_Attendant_PAX_Call_Visual || Cabin_Attendant_PAX_Call_Audible || PAX_Caller
7   index ← input.getID
8   if M_Response_Actorsindex == empty && input.getStatus == “inactive” then
9     | M_Response_Actorsindex ← input
10  else
11    | M_Validation_Information ← validation information E_Stop ← true.failed
12  if M_Response_Actors filled successfully then
13    | M_Response_Actors ← reset empty
    | Output: Transition B to A1_TRIGGER_PAX_CALL

```

After successful validation of preconditions, a set of actor stimuli sequences and expected system response activities are executed. State 2.3) is used to initiate a single active PAX Call. To do so, a stimulus is prepared by altering the respective data model member of the actor of type *PAX_Caller* stored within the scenario configuration memory. This stimulus is send to the respective actors of the associated EOSS model.

Algorithm 2.3): State A1_TRIGGER_PAX_CALL

```

// reset timer Event
1 E_Stop ← timer Time_Out
  // prepare new stimulus by changing actor data structure values
2 M_Scenario_Configuration ← set PAX_Call_Request = call
  // sent stimulus to respective actor
  Output: PAX_Caller ← M_Scenario_Configuration
  Output: e Transition to R1_PAX_CALL_INDICATIONS_ON

```

Subsequently, a synchronous transition is fired in order to enter state 2.4). This state is used to determine and evaluate SuD responses to a previously triggered

PAX Call activation. System responses are expected in terms of visual and audible PAX Call indications for cabin crew members. Moreover, visual PAX Call indication is expected by the respective passenger that activated the call. All indications need to be active in order to constitute a valid system response (intended behavior). In addition, visual crew indications shall provide information on the position of current calls (intended quality). Indicated call positions are also compared to the *PAX_Caller* position data received during state 2.1).

Algorithm 2.4): R1_PAX_CALL_INDICATIONS_ON

```

// current PAX Caller info
1 activeCaller ← M_Scenario_Configuration
  // wait for EOSS responses for visual and audible indications
2 while Input: Cabin_Attendant_PAX_Call_Visual || Cabin_Attendant_PAX_Call_Audible || PAX_Caller
3 do
4   inputCABvisual ← Cabin_Attendant_PAX_Call_Visual
5   indexCABvisual ← inputCAB.getID
6   inputCABaudible ← Cabin_Attendant_PAX_Call_Audible
7   indexCABaudible ← inputCAB.getID
8   inputPAX ← PAX_Caller
9   indexPAX ← inputPAX.getID
  // evaluate functional requirements for each response
  // necessary HMI indications ON?
10  if M_Response_ActorsindexCABvisual == empty && inputCABvisual.getStatus == "active" &&
    inputCABvisual.getPAXCallPosition == activeCaller.getPAXCallPosition then
11    [ M_Response_ActorsindexCABvisual ← inputCABvisual
12  else if M_Response_ActorsindexPAX == empty && inputPAX.getID == activeCaller.getID &&
    inputPAX.getStatus == "active" then
13    [ M_Response_ActorsindexPAX ← inputPAX
14  else if M_Response_ActorsindexCABaudible == empty && inputCABaudible.getStatus == "active" then
    // evaluate non-functional requirement for each response
    // audio volume within required range?
15    if inputCABaudible.getVolume == within defined range then
16      [ M_Response_ActorsindexCABaudible ← inputCABaudible
    // after becoming active, chimes shall automatically be deactivated (fixed chime duration)
17  else if M_Response_ActorsindexCABaudible != empty && M_Response_ActorsindexCABaudible.getStatus ==
    "active" && inputCABaudible.getStatus == "inactive" then
18    [ M_Response_ActorsindexCABaudible ← inputCABaudible
19  else
20    [ M_Validation_Information ← validation information E_Stop ← true.failed
21  if M_Response_Actors filled successfully && all necessary responses received then
22    [ M_Response_Actors ← reset empty
    [ Output: Transition C to A2_WAIT_FOR_HELP

```

Audible indication is expected to be provided by a chime-like functionality. Chime responses are also evaluated with respect to quality property *volume*. Therefore, the observed chime volume is compared to the range provided by the respective quality objective requirement determined by parameter *Chime_Volume*. As chimes have a limited playback time, a chime deactivation is expected sometime after chime activation. If all responses have been received and evaluated correctly, transition C is fired to reach state 2.5).

Although state 2.5) represents a second stimuli sequence activity, no actual stimulus is created for the cabin management system. Instead, a timer is set with a uniformly distributed random amount of time. The specific timer value is located between lower and upper bound of the objective provided by parameter *Wait_Objective*. This time period is used to symbolize a variable amount of time that is expected to pass before a cabin crew member is able to help a passenger. After the timer has been set, a synchronous transition is fired to change to state 2.6).

Algorithm 2.5): State A2_WAIT_FOR_HELP

```

// wait for a random amount of time within range of Wait_Objective
// uses timer Event
1 time_to_wait ← timer random(Wait_Objective.min, Wait_Objective.max)
2 E_Timer ← time_to_wait
   // make sure timeout timer Event is longer than time_to_wait
3 E_Stop ← timer Time_Out + time_to_wait
Output: e Transition to R2_NO_CHANGES

```

State 2.6) is used to observe potential SuD state changes and responses. Only if no responses occur at all and the previously defined random amount of time to wait has passed, transition D is fired to reach state 2.7).

Algorithm 2.6): State R2_NO_CHANGES

```

// observe EOSS system responses
// no new responses from associate EOSS actors shall occur!
1 while Input: Cabin_Attendant_PAX_Call_Visual || Cabin_Attendant_PAX_Call_Audible || PAX_Caller
2 do
3   | M_Validation_Information ← validation information
4   | E_Stop ← true.failed
5 if E_Timer == true then
Output: Transition D to A3_PAX_CALL_RESET
6

```

During state 2.7), a new stimulus is prepared and sent to the associated EOSS model in order to reset the currently active PAX call. This is done similar to state 2.3).

Algorithm 2.7): State A3_PAX_CALL_RESET

```

// reset timer Event
1 E_Stop ← timer Time_Out
   // prepare new stimulus by changing actor data structure values
2 M_Scenario_Configuration ← set PAX_Call_Request = reset
   // sent stimulus to respective actor
Output: PAX_Caller ← M_Scenario_Configuration
Output: e Transition to R3_PAX_CALL_INDICATIONS_OFF

```

At the end of state 2.7), a synchronous transition is fired to reach state 2.8). This state is used to determine and evaluate SuD responses to an active PAX Call that has just been reset. System responses are expected in the form of visual indications to cabin crew members and to the respective passenger that reset the call. This time, no audible indication shall be provided. All indications need to be inactive in order to constitute a valid system response. If all responses have been received and evaluated correctly, transition E is fired to enter state 2.9).

After successful completion of activities for actor stimuli sequences and expected SuD responses, different postconditions are determined and evaluated during state 2.9). At the end of each PAX Call scenario run, a check is performed to ensure, that no PAX Call is currently active. In order to do so, current states of all actors *Cabin_Attendant_PAX_Call_Visual*, *Cabin_Attendant_PAX_Call_Audible* and *PAX_Caller* are requested from the associated EOSS model and stored upon reception. Each EOSS actor shall provide exactly one status response with indication status “inactive”. If all responses have been received successfully, a check is performed to determine if all scenario configuration possibilities have been performed successfully. In case all possible configurations have already been performed and

Algorithm 2.8): R3_PAX_CALL_INDICATIONS_OFF

```

// current PAX Caller info
1 activeCaller ← M_Scenario_Configuration
  // no EOSS responses for audible indications shall occur
2 while Input: Cabin_Attendant_PAX_Call_Audible
3 do
4   M_Validation_Information ← validation information
5   E_Stop ← true.failed
  // wait for EOSS responses for visual indications
6 while Input: Cabin_Attendant_PAX_Call_Visual || PAX_Caller
7 do
8   inputCABvisual ← Cabin_Attendant_PAX_Call_Visual
9   indexCABvisual ← inputCAB.getID
10  inputPAX ← PAX_Caller
11  indexPAX ← inputPAX.getID
12  if M_Response_ActorsindexCABvisual == empty && inputCAB.getStatus == "inactive" then
13    M_Response_ActorsindexCABvisual ← inputCABvisual
14  else if M_Response_ActorsindexPAX == empty && inputPAX.getID == activeCaller.getID &&
15    inputPAX.getStatus == "inactive" then
16    M_Response_ActorsindexPAX ← inputPAX
17  else
18    M_Validation_Information ← validation information E_Stop ← true.failed
19  if M_Response_Actors filled successfully && all necessary responses received then
    M_Response_Actors ← reset empty
    Output: Transition E to POST_CONDITIONS

```

evaluated, the overall timeout event is set in order to indicate successful completion of overall state 2). Otherwise, transition F is fired to reach state 2.10) which initiates a consecutive scenario execution run with altered scenario or customization configurations.

Algorithm 2.9): State POST_CONDITIONS

```

// reset timer Event
1 E_Stop ← timer Time_Out
  // check that no PAX Call is currently active
Output: Cabin_Attendant_PAX_Call_Visual ← request current status of
  Cabin_Attendant_PAX_Call_Visual1
2 for i = 1 to 3 do
  Output: Cabin_Attendant_PAX_Call_Audible ← request current status of
  Cabin_Attendant_PAX_Call_Audiblei
3 for i = 1 to 36 do
  Output: PAX_Caller ← request current status of PAX_Calleri
4 while Input: Cabin_Attendant_PAX_Call_Visual || Cabin_Attendant_PAX_Call_Audible || PAX_Caller
5 do
6   input ← Cabin_Attendant_PAX_Call_Visual || Cabin_Attendant_PAX_Call_Audible || PAX_Caller
7   index ← input.getID
8   if M_Response_Actorsindex == empty && input.getStatus == "inactive" then
9     M_Response_Actorsindex ← input
10  else
11    M_Validation_Information ← validation information E_Stop ← true.failed
  // initiate next run of scenario validation process?
12  if M_Response_Actors filled successfully && M_Reconfiguration_Status != done then
13    M_Validation_Information ← validation information && M_Response_Actors ← reset empty
    Output: Transition F to PARAMETER_RECONFIGURATION
  // scenario validation process completed successfully
14  else if M_Response_Actors filled successfully && M_Reconfiguration_Status == done then
15    M_Validation_Information ← validation information && E_Stop ← true.success

```

State 2.10) is used to modify and recombine scenario and customization parameter configurations. This state also determines the number of consecutive runs for

scenario PAX Call. In this example, scenario parameter configuration is used to determine different actors of type *PAX_Caller* similar to state 2.1) for each run. This is done in order to ensure that the scenario is valid for all 36 actors of type *PAX_Caller*.

In addition, each useful customization parameter combination shall be evaluated (cf. customization model Figure 5.12). This includes combinations for different indication colors and modes. The overall number of useful customization parameter combinations is 30 from a set of 54 possible combinations. This is because indication mode “steady” does not require validation of different blink characteristics. As part of this example, 3 parameter combinations are used together with 36 consecutive runs for each possible actor of type *PAX_Caller*. Thus, the example scenario is executed 108 times before completion.

Algorithm 2.10): State PARAMETER_RECONFIGURATION

```

// reset timer Event
1 E_Stop ← timer Time_Out
// prepare new scenario configuration?
2 if M_Reconfiguration_Status != all scenario configuration combinations completed then
  // set next PAX Caller actor
3   M_Scenario_Configuration ← next actor of type PAX_Caller
// prepare new customization model configuration?
4 else if M_Reconfiguration_Status != all customization configuration combinations completed &&
  M_Reconfiguration_Status == all scenario configuration combinations completed then
5   M_Reconfiguration_Status ← reset scenario configuration, PAX_Caller1
   M_Customization_Configuration ← next possible customization configuration
   Output: A_PAX_Call_Customization ← M_Customization_Configuration
// customization model configuration complete and last scenario configuration?
6 else
7   M_Reconfiguration_Status ← done
// after each reconfiguration, begin new scenario cycle
Output: e Transition to PRECONDITIONS

```

State 3) represents the final active state after successful or unsuccessful completion of the overall scenario. When entering this state from state 2), final validation information is written into memory *Validation_Information*. Moreover, a control flow token is produced at the output port *d* while a synchronous transition is executed that will put the overall scenario back into state 1). After that, the overall scenario could be re-triggered and executed again.

Algorithm 3): State FINISHED

```

1 M_Validation_Information ← final validation information
Output: output port d ← control flow token (1 == success; 0 = failure)
Output: e Transition to IDLE

```

5.4.6 HMI Concept Model Development for CMS

Based on the extensive market and technology trends analysis from concept development and in collaboration with other stakeholders, i.e. cabin management system (CMS) experts, human machine interface (HMI) concepts were developed in the form of graphical user interfaces (GUIs). Moreover, atomic mission, service and scenario models were used as input, e.g. in order to determine possible HMI users from the overall set of available actors or to determine necessary input/output data requirements for HMI models. For the basic graphical design with associated *look*

and feel, ARINC standard 837 - design guidelines for aircraft cabin HMIs [86] as well as preliminary graphical design studies from concept design, e.g. in the form of *MS Visio* sketches or drawing made with the free software *Gimp* were used.

Reconsider the example use case “*PAX Call*”. An HMI design for a suitable passenger interface supporting this use case could have different characteristics. Figure 5.20 shows four different graphical HMI design concepts. Three different concepts for single cabin passenger interfaces are shown (top, left to right) that can be used to call cabin attendants. Concepts (1) and (2) represent dedicated HMIs for “*PAX Call*” while concept (3) is a combined multi-use-case HMI that includes additional interface elements for service “*Provide Personal Reading Lights*”. At the bottom of Figure 5.20, another combined passenger service interface concept (4) is depicted that is intended to be used by two passengers and unites services “*PAX Call*”, “*Provide Personal Reading Lights*”, “*Provide Fasten Seatbelt Indication*”, and “*Provide No-Smoking Indication*”.

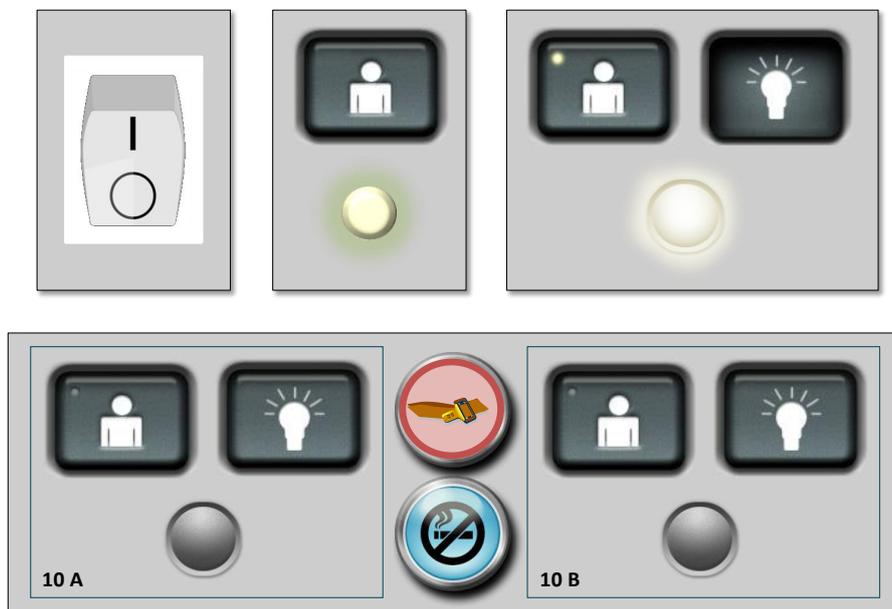


Figure 5.20 – Different human machine interface design concepts for a single passenger interface (top) to call cabin attendants (1) and (2), a combined interface that includes reading light control (3) as well as a combined multi-user interface with additional fasten seatbelt and no smoking allowed indication (bottom)

Concept (1) represents a switch that has two positions, representing on and off. In this case, the current status of the interface is directly visible at any time by evaluating the current switch position. This design solution does not necessarily require any additional visual feedback information for an actor who triggers a call, e.g. an additional indication light. Thus, the data model shown in Figure 5.9 of chapter 5.4.2 would not necessarily need a data structure *PAX_Call_Indication* with corresponding members *active* and *inactive* since no active system response is required. When choosing this HMI concept, scenarios for the respective use case need to be adjusted in order to not expect visual *PAX Call* indication responses from the respective passenger actor (HMI concept model feedback).

In contrast to variant (1), variants (2) and (3) shown in Figure 5.20 use active indication lights in order to provide status information to passengers, whereby variant

(2) uses a push button with dedicated signal light and variant (3) provides a push button with integrated indication. In both cases, indication lights are changed actively in response to passenger induced HMI input triggers. Active feedback for PAX Calls may be required in general if considering another use case that affects the operation of this specific HMI, called “*Inhibit PAX Calls*”. As part of this use case, cabin crew members may actively inhibit the activation of calls. “*PAX Call*” inhibition may also occur automatically, e.g. during different flight phases. In both cases, no signal lights or any other indication shall be activated.

In addition to the *PAX Call* status indication function, both HMI variants (2) and (3) provide an additional advantage. When using an indication light instead of a toggle switch, active *PAX Calls* are more easy to spot by cabin attendants. Since variant (4) at the bottom of Figure 5.20 also represents a multi-user interface, additional seat labels and boundary markings are provided to help users choosing the right interface. The above examples illustrate, that it is hard to include and evaluate usability requirements without the creation of an HMI concept model or virtual prototypes. Moreover, decisions from HMI concept development may have a considerable impact on atomic mission, service and scenario model development. Thus, early HMI concept modeling is crucial for minimizing product uncertainty and for decreasing the overall design space early during development.

In a second step of HMI concept development, a student at the *Hamburg University of Applied Sciences* was assigned the task of developing executable GUI model components for a basic CMS with MLDesigner. Firstly, possible graphical designs were developed and validated by using tools for conceptual graphics together with Tcl/Tk GUI builder tools including *Visual Tcl* [11], *SpecTcl* and *GUI Builder* [113]. Subsequently, the developed Tcl/Tk scripts were adjusted for MLDesigner and linked with default Tcl/Tk script executing modules (cf. chapter 4.3.6).

Since customization has a huge impact on HMI development, it is important to include customization aspects in the design of CMS HMI and GUI concepts. Consider the example use case “*PAX Call*” with customization model depicted in Figure 5.12 of chapter 5.4.2. In this case, a GUI model component can be developed that provides the means for different PAX Call status indication modes, e.g. signal lights with steady and blinking modes or with different indication light colors. In this case, GUI models dynamically adapt appearance or behavior depending on the current customization model configuration. Moreover, custom graphic designs may be implemented for each airline, e.g. to show a specific logo. However, due to the basic design approach chosen for CMS development, only basic graphic designs were used that are suitable for most stakeholders.

5.4.7 CMS Requirements Specification Validation

In the course of the development of an executable requirements specification (ERS) for a basic CMS, ERS validation was performed based on the validated concept design and stakeholder input. Mission models as well as HMI concept models and service models were validated with the aid of CMS experts of a major aircraft manufacturer, certification documents, and end users. This was done, for instance, by performing design reviews in the form of group meetings [373].

The validation of quality objective models, atomic mission models and scenario models was mainly done based on single model reviews with regard to concept

design and engineering judgment. However, especially in the case of scenario model validation, the participation of other CMS stakeholders was found to be most helpful. In the case of HMI concept model development, early usability validation was also performed with the aid of students of the *Hamburg University of Applied Sciences* at the *Cabin and Cabin Systems Lab* (CCS-Lab) in Hamburg, Germany.

5.5 CMS Executable Overall System Specification

5.5.1 Executable CMS Specification Model

Based on the developed and validated executable requirements specification (VERS) for a basic cabin management system (CMS), an executable overall system specification (EOSS) was developed. Figure 5.21 shows the top-level view of the developed specification model. The system under design (SuD), i.e. the cabin management system, is part of the central module of the overall model (aircraft cabin domain). Models of system environment are arranged around the central SuD module. All top-level model entities are coupled via ports and transitions in order to form an EOSS at extended overall aircraft level.

Environment models cover multiple aircraft domains. Differentiation between different aircraft or external subsystems is made based on the numbered standard classification for commercial aircraft with so-called *ATA chapters*, as defined by the *Air Transport Association* (ATA). The environment model includes ATA 31 cockpit (leftmost box), ATA 77 and 24 power generation (rightmost box), various aircraft subsystems including ATA 32 landing gear, ATA 27 flight control, ATA 21 environmental control / cabin pressure control, ATA 52 aircraft doors / evacuation slides system, ATA 44 in-flight entertainment (upper box) as well as different ground service entities (lower box), e.g. ATA 23 for ground service communication.

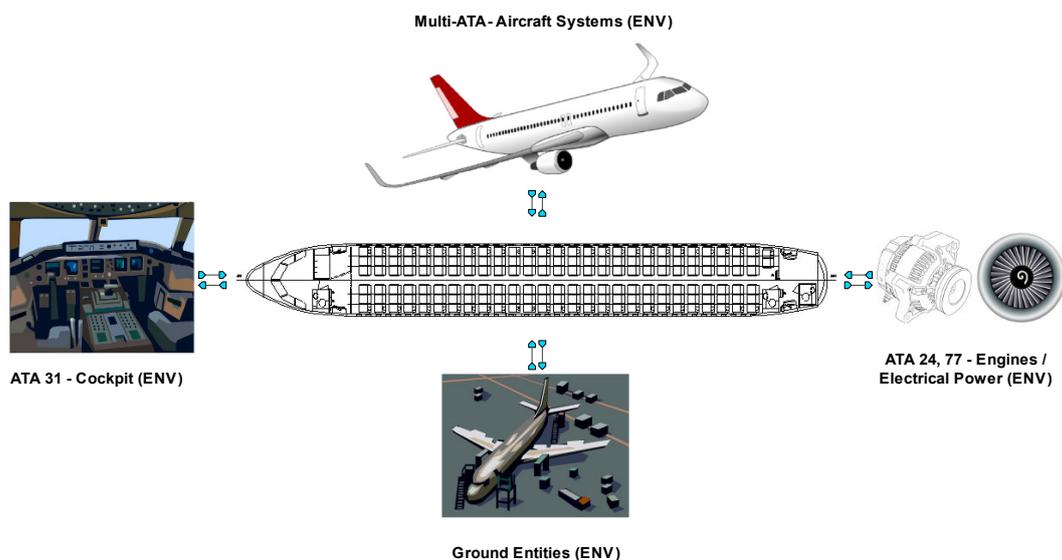


Figure 5.21 – Top level of the executable overall system specification for a basic cabin management system (central module) in MLDesigner

Environment models contain abstract sub-models of functional and non-functional design properties that are necessary to operate the SuD. For example, a basic power

generation system is required to operate all other electronic system models. During mission model and overall system model execution, system and system environment states are changed, thus changing the coupled state of the overall system. System actors from VERS development are included at necessary interaction points, e.g. human actors representing ground service personnel or non-human actors, e.g. to observe flight mode states. Moreover, non-functional observer modules are included in order to observe non-functional design properties, including overall system weight, cost and dynamical power consumption.

The developed CMS is integrated part of the cabin core systems domain and has links to different other aircraft subsystems. Figure 5.22 shows the SuD at system level. With regard to the aim of creating an interactive system demonstrator, i.e. a virtual prototype, the CMS model was limited in number of equipment for the architecture model. The CMS design depicted in Figure 5.22 was created based on an existing setup of a physical aircraft cabin mock-up that is installed in the *Cabin and Cabin Systems Lab* (CCS-Lab) of the *Hamburg University of Applied Sciences* in Hamburg, Germany (cf. reference [373]). Because of the limited amount of space, redundancy may not be compliant with real aircraft requirements for safety.

The architecture model shown in Figure 5.22 is part of the rear section of a single aisle aircraft cabin. It consists of different subsystems and elements (boxes) that are connected through different networks (black boxes at the top and at the bottom) and includes power provisioning from upper level system environment models (blue lines). Networks are also used to connect the CMS to other avionics systems, the cockpit as well as to ground service facilities when on ground and activated. Moreover, the SuD model includes system actors at specific interaction points (boxes with actor symbol). Actors are also included within element modules, e.g. passenger service units (PSU), in order to decrease graphical complexity (cf. Figure 5.23). Two lavatory (LAV) components, which are not part of the CMS but CMS environment, are integrated within the SuD model, since they contain different CMS elements and functions, e.g. passenger indication signs or audio sinks for cabin announcements. A central control computer is used to coordinate all multi-user related CMS functions (left-hand side). Four multi-purpose handset stations exist for a range of communication and safety functions, e.g. aircraft intercommunication, ground service communication or public address. Other cabin crew actor related CMS elements include a multi-purpose information and control panel (middle section) as well as three dedicated cabin speakers. Six seat rows exist that provide three seats at each side of the cabin aisle. Each seat row is equipped with a PSU module that contains a set of sub-elements such as passenger-related speakers, reading lights, indication panels as well as cabin attendant call panels. In total, the following number of CMS model elements is used: 1 control computer, 3 global cabin speakers, 4 handsets, 1 multi-purpose indication panel, 2 lavatory monuments, 3 different networks and 12 PSUs.

The overall architecture model is decomposed hierarchically into elements (E), sub-elements and functions (F) as depicted in Figure 5.23, using the example of a PSU (top) with public address function statechart (bottom). Each lower lever of abstraction increases the degree of design detail. Functions, that are modeled in the form discrete event models and statecharts, are allocated to elements that represent a specific design solution or concept. It is also possible to model functions with custom source code primitives or as a combination of basic library modules.

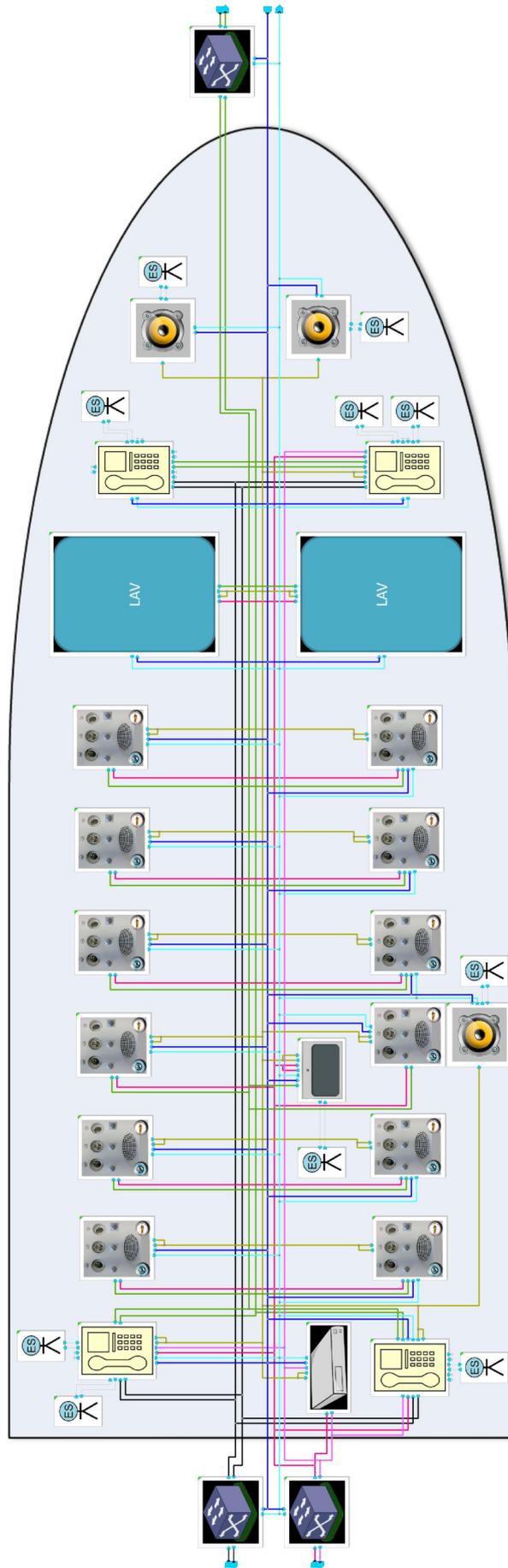


Figure 5.22 – System level view of an executable specification for a basic cabin management system in MLDesigner

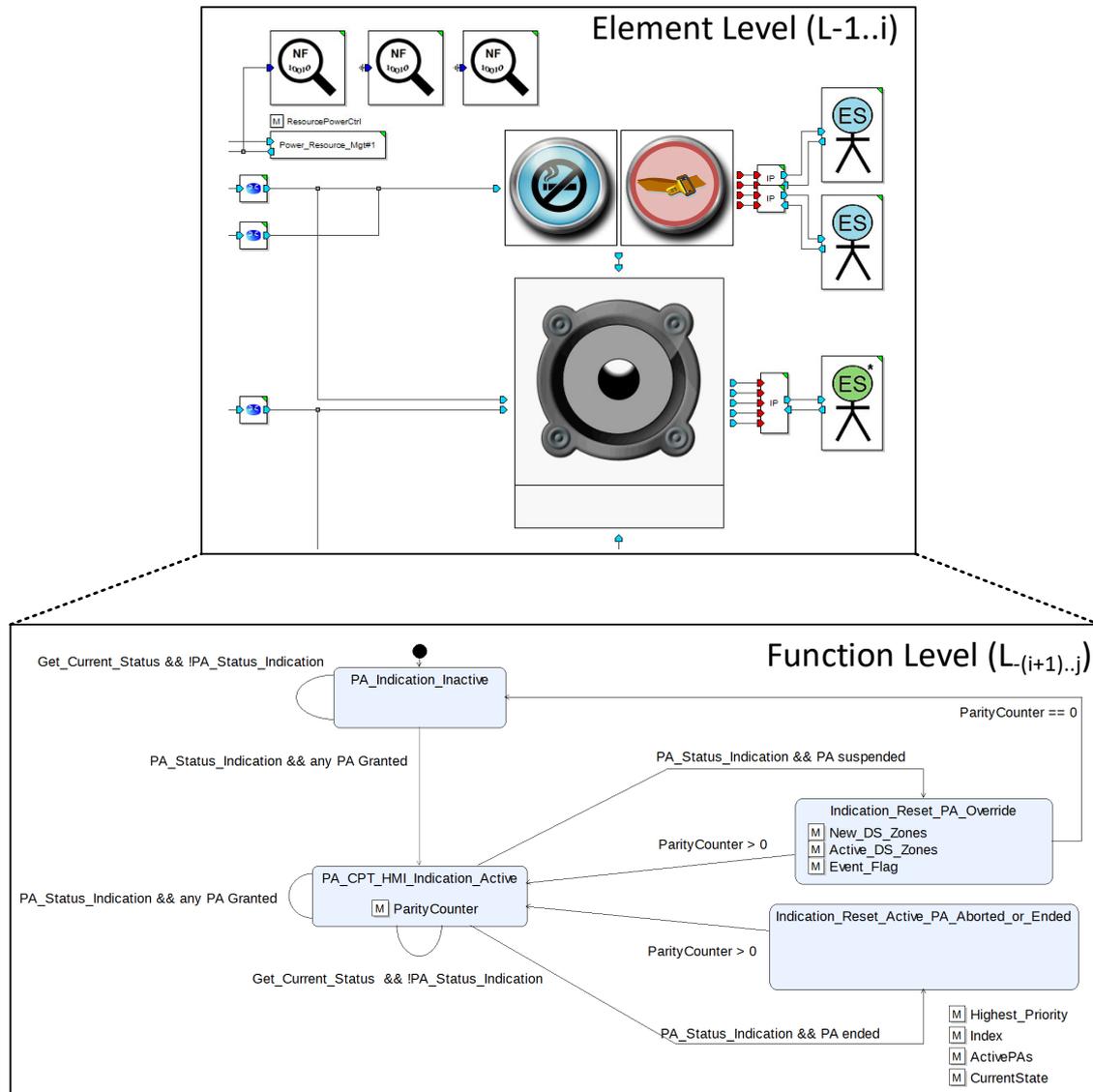


Figure 5.23 – General hierarchical structure of elements, sub-elements and functions of a cabin management system in MLDesigner, using the example of a passenger service unit (top) with public address function block (bottom)

The following CMS architecture model structure with associated elements, functional allocations (high level functions only) and non-functional parameter objective specifications {upper bound, mean/target, lower bound} was used:

- (E) Passenger Service Unit (PSU) [total number: 12]
 - Overall Element Cost Objective (US \$): {700,1000,1200}
 - Overall Element Weight Objective (kg): {1,1.5,2}
 - Overall Element Power Objective (watts): {100,105,120}
 - (E) Passenger Notification Panel [total number: 1]
 - * (F) Fasten Seatbelt Indication Function
 - * (F) Smoking Allowance Indication Function
 - (E) PSU Speaker [total number: 1]

- * (F) Chime Function
 - (F) Audible Passenger Status Notification Function
 - (F) Audible Cabin Crew Status Notification Function
- * (F) Public Address Sink Function
- (E) Cabin Attendant Call Panel [total number: 1]
 - * (F) Cabin Attendant Call Function
 - * (F) Cabin Attendant Call Indication Function
- (E) Passenger Reading Light Panel [total number: 1]
 - * (F) Passenger Reading Light Function
- (E) Handset [total number: 4]
 - Overall Element Cost Objective (US \$): {2000,3000,4000}
 - Overall Element Weight Objective (kg): {1,2,4}
 - Overall Element Power Objective (watts): {120,145,200}
 - (F) Cabin to Cabin Intercommunication Function
 - (F) Cabin to Cockpit Intercommunication Function
 - (F) Cabin to Ground Service Intercommunication Function
 - (F) Public Address Source Function
- (E) Multi-Purpose Information and Control Panel [total number: 1]
 - Overall Element Cost Objective (US \$): {500,1000,1500}
 - Overall Element Weight Objective (kg): {2,3,5}
 - Overall Element Power Objective (watts): {60,80,120}
 - (F) Fasten Seatbelt Indication Function
 - (F) Smoking Allowance Indication Function
 - (F) Cabin Crew Reading Lighting Control Function
 - (F) Cabin Attendant Call Indication Function
 - (F) Cabin Attendant Call Control Function
 - (F) Aircraft Doors and Slides Indication Function
 - (F) Cabin Ready Control Function
- (E) Lavatory Component [total number: 2]
 - Overall Element Cost Objective (US \$): {1000,1500,2000}
 - Overall Element Weight Objective (kg): {5,10,15}
 - Overall Element Power Objective (watts): {50,60,70}
 - (E) Passenger Notification Panel [total number: 1]
 - * (F) Return to Seat Indication Function
 - * (F) Smoking Allowance Indication Function
 - (E) LAV Speaker [total number: 1]

- * (F) Chime Function
 - (F) Audible Passenger Status Notification Function
 - (F) Audible Cabin Crew Status Notification Function
- * (F) Public Address Sink Function
- (E) Cabin Attendant Call Panel [total number: 1]
 - * (F) Cabin Attendant Call Function
 - * (F) Cabin Attendant Call Indication Function
- (E) Central Control Computer [total number: 1]
 - Overall Element Cost Objective (US \$): {2000,4000,6000}
 - Overall Element Weight Objective (kg): {30,40,50}
 - Overall Element Power Objective (watts): {320,350,400}
 - (F) Public Address Management Function
 - (F) Reading Light Management Function
 - (F) Cabin Attendant Call Management Function
 - (F) Crew and Passenger Notification Management Function
- (E) Cabin Crew Speaker [total number: 3]
 - Overall Element Cost Objective (US \$): {500,1700,2200}
 - Overall Element Weight Objective (kg): {2,3,5}
 - Overall Element Power Objective (watts): {100,150,200}
 - (F) Chime Function
 - * (F) Audible Passenger Status Notification Function
 - * (F) Audible Cabin Crew Status Notification Function
- (E) Aircraft Systems Network [total number: 1]
 - Overall Element Cost Objective (US \$): {1000,2000,4500}
 - Overall Element Weight Objective (kg): {50,60,70}
 - Overall Element Power Objective (watts): {70,100,120}
 - (F) Data Routing Function
- (E) Cockpit/Cabin Network [total number: 1]
 - Overall Element Cost Objective (US \$): {1000,2000,4500}
 - Overall Element Weight Objective (kg): {50,60,70}
 - Overall Element Power Objective (watts): {70,100,120}
 - (F) Data Routing Function
- (E) Ground Station Network [total number: 1]
 - Overall Element Cost Objective (US \$): {1000,2000,4500}
 - Overall Element Weight Objective (kg): {50,60,70}

- Overall Element Power Objective (watts): {70,100,120}
- (F) Data Routing Function

In the course of CMS development, the use of functions in the form of combined discrete event models and statecharts was found to be most useful in order to identify formal design flaws within system models [373]. For instance, to identify incomplete state transitions as depicted in Figure 5.24 for the example of a cabin doors and slides status indication function. Adjustments made after formal design evaluation are marked in red. Figure 5.24 includes corrections made for missing transitions (transition from state *Idle* to state *Doors_and_Slides_OK*) and missing guard conditions, e.g. in case state changes that shall only occur in case no *Reset* event was triggered.

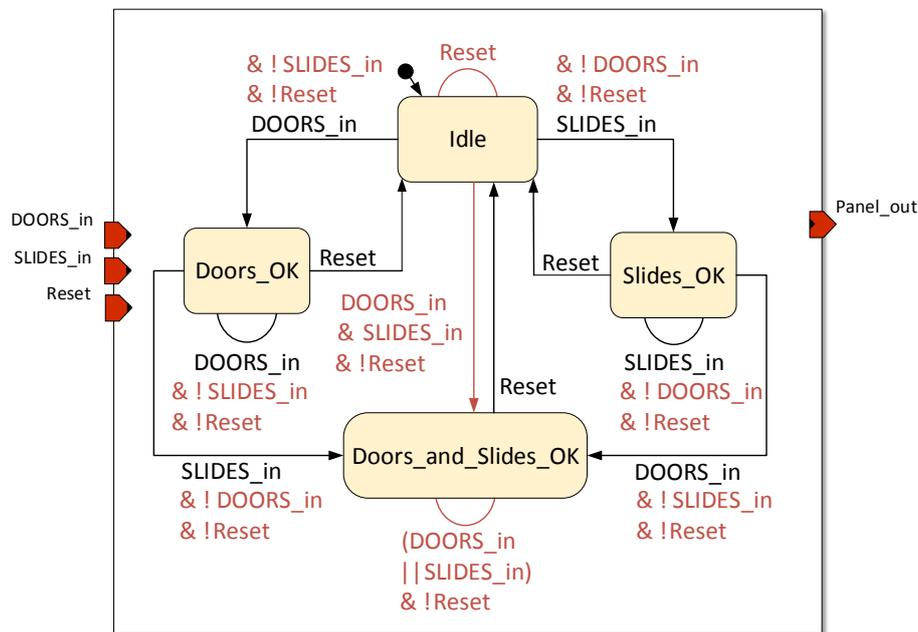


Figure 5.24 – Function statechart for cabin doors and slides status indication with corrected information (red) after formal design evaluation (Source: Wiegmann 2014 [373])

In parallel to the development of an EOSS model, the mind map from concept design (cf. chapter 5.3) was extended in order to document system model development. At the end of the development project, around 1200 mind map nodes were created, divided over ten levels of hierarchy [373].

5.5.2 Automated CMS Specification Validation

After the development of an executable overall system specification (EOSS) for a basic cabin management system (CMS), an executable validation flow (EVF) was developed. As depicted in Figure 5.25, the EVF includes the EOSS and the validated executable requirements specification (VERS) (cf. chapter 5.4.1) as well as a predefined validation report generation module (cf. chapter 4.4.2).

During the last stage of the CMS specification development, a sequence of EVF executions was performed in order to validate the EOSS. During this process, necessary specification adjustments of the EOSS were made based on the evaluation

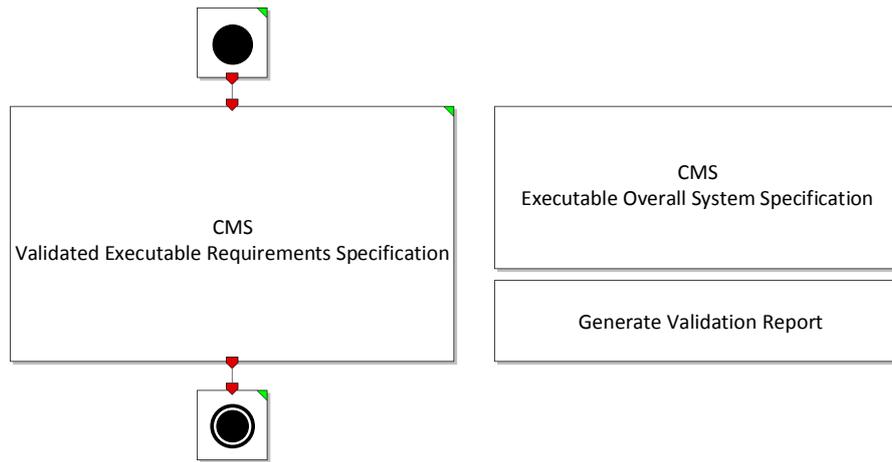


Figure 5.25 – Executable validation flow for a basic cabin management system in MLDesigner

of the automatically created validation report. Based on the mission model described in chapter 5.4.1, three different quality objectives (QO) were validated (cf. chapter 5.4.3) together with six different atomic mission (AMIS) models. Table 5.1 shows quality objectives $QO1$ (overall system weight in kg), $QO2$ (overall system cost in US dollar) and $QO3$ (overall system power consumption in watts):

	QO1	QO2	QO3
Minimum Value	0kg	0\$	100W
Mean / Target Value	400kg	20000\$	2000W
Maximum Value	600kg	80000\$	7000W

Table 5.1 – Quality objectives (QO) for a basic cabin management system

For each of the atomic mission models used in this example, a number of different scenario configurations was executed. As depicted in Table 5.2, a total of 1823 different scenarios were executed as part of the overall mission model during one EVF simulation run.

ID	Atomic Mission Model	Number of Executed Scenario Configurations
1	Cabin to Ground Service Intercommunication	1
2	Cabin to Cabin / Cockpit to Cabin Intercommunication	1
3	Public Address (PA)	1677
4	Passenger Calls for Help (PAX Call)	108
5	Lavatory Smoke Indication (LAV Smoke)	4
6	Automated Fasten Seatbelt (FSB) Indication	16
7	Automated No Smoking (NS) Indication	16

Table 5.2 – Number of executed atomic mission models with associated scenario configurations for a basic cabin management system

The VERS was successfully used to automatically validate all relevant functional and non-functional properties of the EOSS model. At the end of EVF execution, two types of validation reports were generated. A textual report as well as a report in tabular format. Both validation reports can also be used to perform different post simulation analysis, e.g. by using spreadsheet software. Post simulation analysis enables system architects to analyze non-functional aspects of system architecture

more closely. Both of the automatically generated validation reports (textual and tabular format) are provided unabridged in appendix B.4 and appendix B.5.

Table 5.3 shows results of the generated validation reports for non-functional design properties (NFP) in relation to non-functional design objectives. For static quality objective QO1, the values for NFP1 that were determined during EVF execution indicate, that the current overall system weight is well below the maximum and still below the expected target / mean value of the quality objective. The values of NFP2, which is related to QO2, are all above the target / mean value of the quality objective but still within the limitations provided by the objective maximum.

In contrast to the static quality objectives QO1 and QO2, QO3 is a dynamic objective. This means, that the values of NFP3 change dynamically during overall system execution and are strongly dependent on the overall mission model. System properties for dynamic objectives can thus not be calculated statically at simulation start-up but are determined dynamically for each change during EVF execution. Based on the results of Table 5.3, it may be concluded that overall power consumption is well between the limits of the given objective. This is because target and minimum value of NFP3 are close to the initial expectations determined by QO3.

	QO1	QO2	QO3	NFP1	NFP2	NFP3
Minimum Value	0kg	0\$	100W	214kg	25400\$	115W
Mean Value	400kg	20000\$	2000W	278kg	43100\$	1690W
Maximum Value	600kg	80000\$	7000W	350kg	62000\$	3592W

Table 5.3 – Comparison of quality objectives (QO) and determined non-functional design properties (NFP) for a basic cabin management system

In order to execute the EVF and to evaluate validation results, mobile computer platforms were used together with *MLDesigner* version 3.0.0.1041 for *Microsoft Windows* 64-bit operating systems. Two different simulation platforms, specified within Table 5.4, were used for the development of a basic CMS, including automated validation. At EVF startup, all models are compiled. This takes approximately 2.42 minutes on either of the simulation platforms shown in Table 5.4. Model compilation is only required once during a set of consecutive EVF simulations and each time model changes have been made. After model compilation, a complete EVF execution run needs an average of 26.5 seconds to finish for the example used (mean value calculation based on the performance of 100 consecutive simulation runs on each of the simulation platforms shown in Table 5.4).

	Dell Vostro 3750	Dell XPS L502X
Processor	Intel i7-2630QM 4x 2.0 GHz	Intel i7-2720QM 4x 2.20 GHz
RAM	8GB (1.333 MHz DDR3)	8GB (1.333 MHz DDR3)
Hard-Disk	500GB (SATA, 7200rpm, 16MB cache)	640GB (SATA, 7200rpm, 16MB cache)
Graphics	NVIDIA GeForce GT 425M (1GB)	NVIDIA GeForce GT 540M GR (2GB)

Table 5.4 – Specifications of the simulation platforms used for CMS modeling and execution

5.5.3 CMS Virtual Prototype

For the development of a virtual prototype (VP), graphical user interface (GUI) modules that were developed during human machine interface (HMI) concept design

were integrated within the executable overall system specification (EOSS) model for a basic cabin management system (CMS). In order to do so, system actor modules were exchanged with GUI model components. With this, it is possible to control and evaluate overall system behavior and non-functional design properties interactively during overall system simulation.

Figure 5.26 depicts three different GUI model components that were developed for service “*Passenger Calls for Help*” (*PAX Call*) as well as other CMS services. Row one shows a passenger-related GUI (left), a visual indication panel for cabin crew (middle) and a cabin speaker (right). At simulation startup, all components are in inactive state. When pushing the PAX Call button of a passenger GUI, a stimulus is prepared and sent to the associated system interaction point (second row). In case each of the GUIs depicted in Figure 5.26 receives certain response messages from the CMS model, all three GUI components will change into an active indication state (bottom row). GUI models can also include user interfaces that are specific to simulation models only for evaluation purposes and are not intended to be realized as the associated GUI models. In the case of the cabin speaker depicted in Figure 5.26, the speaker is highlighted when active and, in addition, provides information on audio characteristics. In contrast, a physical implementation of a cabin speaker would provide audible information only. However, visualizing or recording non-functional design properties is important in order to be able to validate use cases with non-functional properties interactively during overall system simulation.

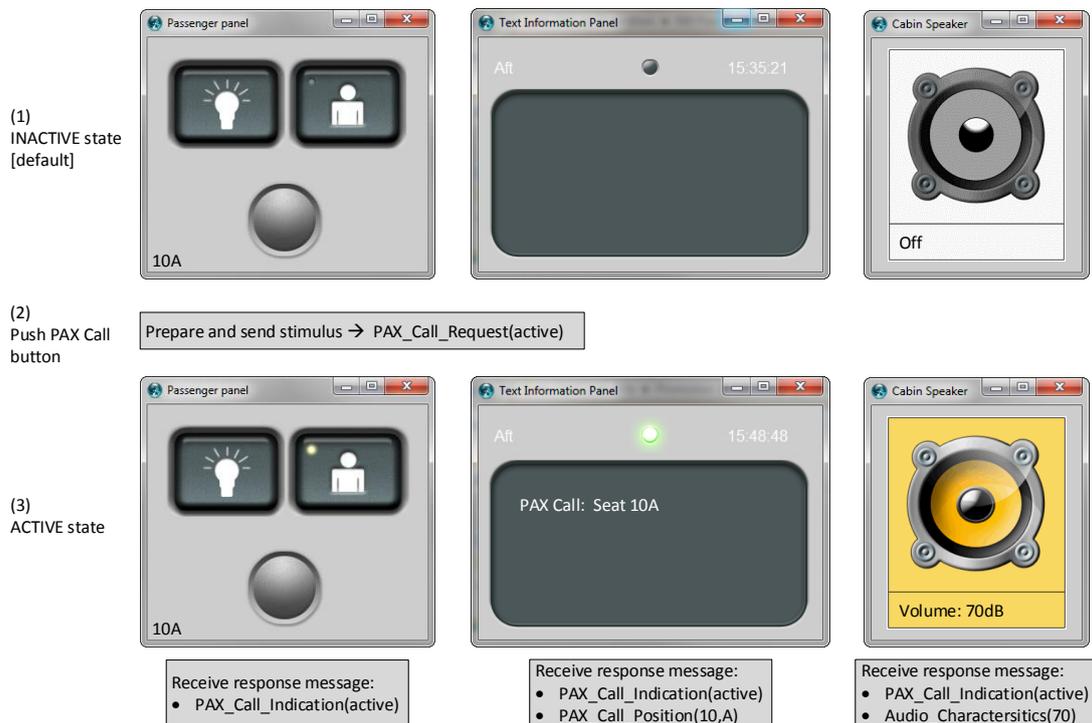


Figure 5.26 – Graphical user interfaces of a virtual CMS prototype for use case “*PAX Call*” during inactive state (top row), activation (middle row) and active state (bottom row)

Consider the example for a use case called “*Passenger Announcement*” or “*Public Address*” (PA). This safety-relevant use case requires different volume levels for different flight phases in order to ensure audibility at all times, e.g. during take-off and landing or during emergencies like rapid loss of cabin pressure. Moreover,

speaker volumes close to a PA source need to be reduced in order to avoid acoustic feedback (also known as *Larsen effect*). Therefore, volume levels are indicated during interactive system model execution. In order to be able to observe system reactions more closely, it is possible to slow down the overall simulation speed by using a global clock event generation module in conjunction with a basic simulation synchronization module in MLDesigner.

As part of another research project, an already existing complex multi-user and multi-use case GUI for cabin management systems should be extended in order to integrate services for aircraft cabin surveillance services [126] and [96]. Figure 5.27 depicts an example view of the central GUI model that was used as part of an EOSS model in MLDesigner. As part of the associated customization concept for cabin surveillance services, an optional customization parameter was determined during interactive validation. This parameter is used to dynamically change the design and behavior of the GUI in a way to indicate location and orientation of cameras within the overall cabin area as shown in Figure 5.27. As a result, data and customization models of related atomic mission models may need to be adapted along with service and scenario models.

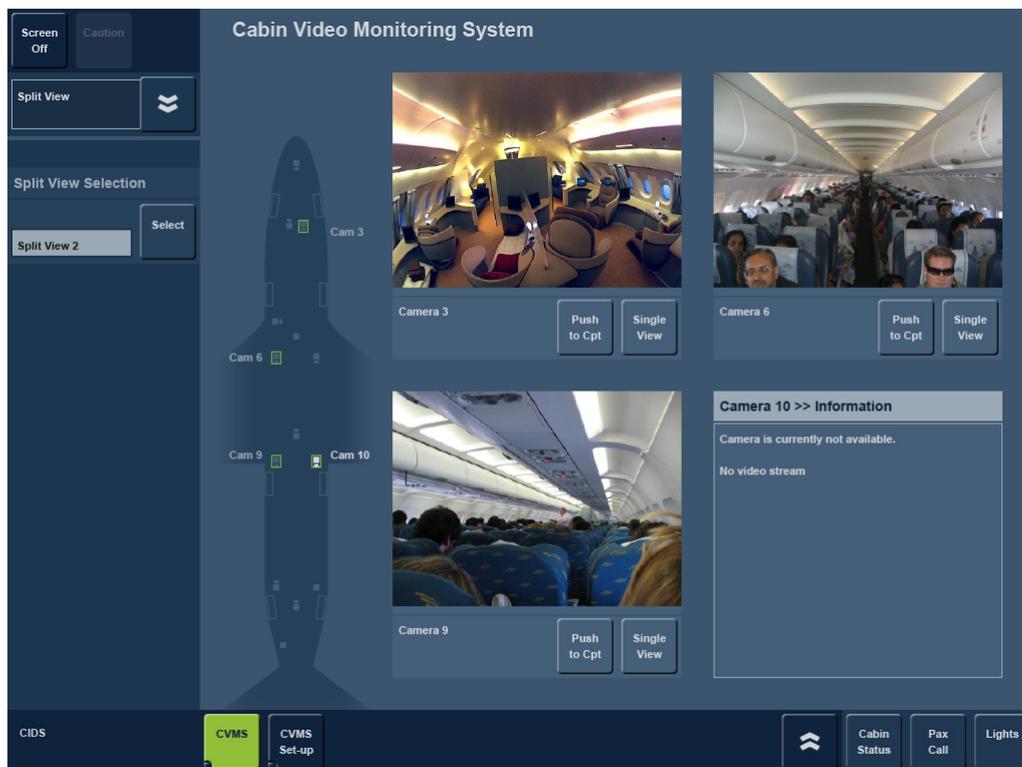


Figure 5.27 – Multi-purpose cabin attendant user interface, designed for multiple users and multiple use cases, e.g. for performing cabin video monitoring services (Source: by courtesy of Airbus Operations GmbH, used for HMI concept design and virtual prototyping [96])

In another example, a CMS system model with associated HMI concept was developed based on already existing textual specifications in order to compare and assess performances of photonic and traditional network architectures (cf. reference [204]). As part of this project, an interactive cabin map GUI was developed for a CMS that shows a bird's eye view of an aircraft cabin as depicted in Figure 5.28. This map is intended to visualize different functional and non-functional information during interactive system simulation, e.g. activated indication lights and overall system

weight. Moreover, each graphical component of the global cabin GUI is designed to be accessible dynamically during simulation by opening dedicated sub-widgets (cf. top of Figure 5.28) for different GUIs upon activation, e.g. passenger service units or flight attendant panels [126]. With regard to cabin and CMS customization, the interactive cabin GUI is designed to be generated according to the configuration of the underlying system specification model. Thus, layout and design of the top-level widget is adapted for different cabin layouts, e.g. by different airlines.

In order to provide GUIs for system and system environment configuration during interactive CMS simulation, different GUIs have been created. Depending on the type of configuration parameter (cf. chapter 4.3.4), different possibilities exist for GUI concept model development. Figure 5.29 depicts some basic configuration GUI examples for a CMS and associated system environment models. Basic quantity parameter data types include integer or float types which can be visualized by interactive input fields (top left), e.g. for aircraft height configuration. The same applies to string type parameters. In order to provide a more limited input provision, predefined buttons (second from top left), e.g. for specific aircraft modes or sliders (third from top left) for current electrical power provision can be used. In the case of enumerated type configuration parameters, either drop-down menu GUIs (bottom left) or radio button GUIs (right) can be used, e.g. for configuring cabin pressure or aircraft flight phases.

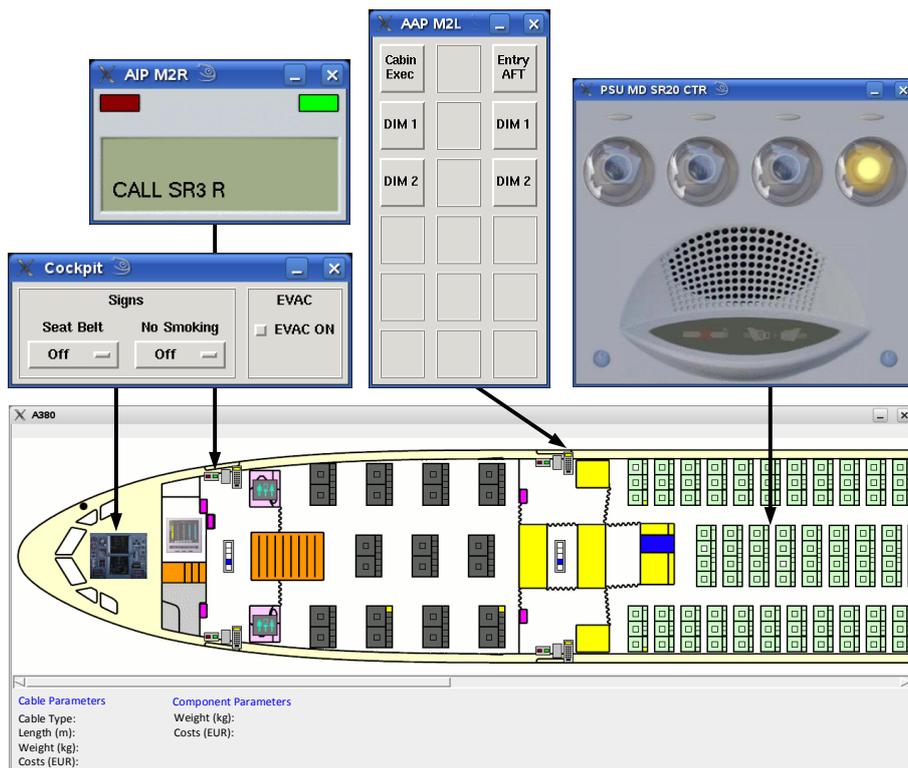


Figure 5.28 – Top-level cabin user interface examples for interactive CMS simulations (adapted from Fischer and Salzwedel 2011 [126])

At the end of the specification development and validation, the virtual CMS prototype was used to create an interactive system mockup for the *Cabin and Cabin Systems Lab* (CCS-Lab) of the *Hamburg University of Applied Sciences* as shown in Figure 5.30.

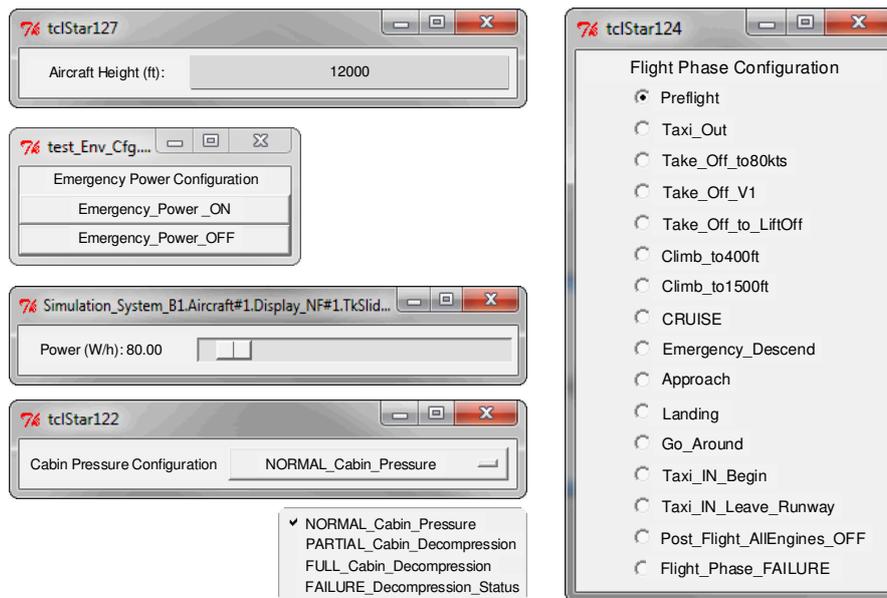


Figure 5.29 – Example Tcl/Tk widgets for changing aircraft system and system environment configuration parameters interactively

In order to do so, the VP is hosted and executed on the simulation platform *Dell Vostro 3750* and operated via different peripheral HMI devices. Both, the VP and CMS mockup can be used to perform additional validation activities with other CMS stakeholders, especially in order to confirm the results of the automated validation process. Therefore, a global validation report is generated during simulation, including 48 sub-reports for each of the interactive validation probes of the overall system model. Moreover, the CMS mockup is intended to be used for further teaching and research activities at the CCS-Lab [373].



Figure 5.30 – Virtual cabin management prototype as part of the *Cabin and Cabin Systems Lab* of the *Hamburg University of Applied Sciences* (Source: Wiegmann 2014 [373])

5.6 Avionics Systems Design Optimization

This section reconsiders the example of avionic system architecture optimization described in references [124], [316] and [317]. By using simulation sets, it is possible to automate the described design and optimization process for distributed avionics at the end of validated executable overall system specification (VEOSS) development.

At the end of chapter 2.3.5, an approach to automate the generation of system architecture models (AM) from behavioral models (BM) is described. A behavioral model can be compared to an executable system specification that is designed without any relation to a specific system architecture, i.e. a design without any network or hardware elements. In contrast, an architecture model uses functional allocation to combine required functionality, system structure and available technology. Thus, an AM is similar to a VEOSS model. The aim of function partitioning and architecture model generation is to provide means to generate different sets of system architecture models based on a common functional model (model generation).

Fischer extended Baumann's automated mapping to optimize cabling for distributed integrated modular avionics (DIMA), [124], [316] and [317]. In reference [124], an AM with optimized DIMA architecture is developed for a cabin pressure control system (CPCS). The general idea is to use an existing validated BM of a CPCS together with given system provisions and constraints in order to find a distribution of functions that will optimize a specific target or a set of targets. Figure 5.31 depicts the basic structure of an AM that is created during system optimization (left-hand side) together with the associated overall system topology (right-hand side). Based on available DIMA installation spaces of the overall system topology, functions (F) are allocated to available sections (S). For network communication between sections and functions, interfaces (I) and channels (C) are generated for a specific type of network (e.g. Ethernet) and linked via the intended network topology.

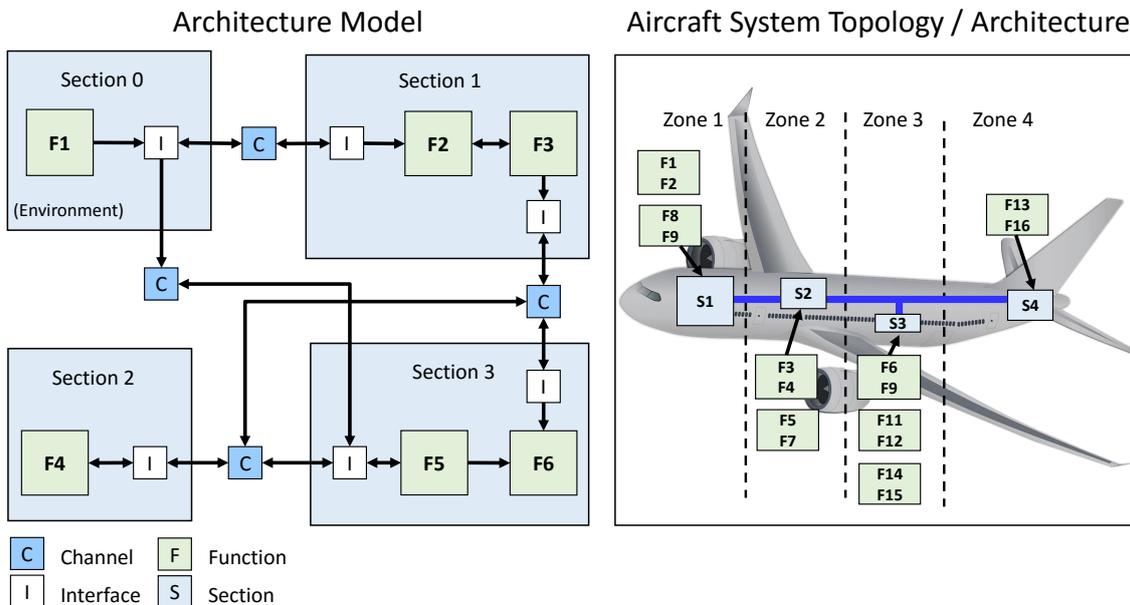


Figure 5.31 – Architecture model with sections, functions, interfaces and channels and overall system topology (adapted from [124] and [316])

Three different models are used as part of this optimization. During a first step, a validated BM is developed for the CPCS.

Secondly, a so-called resource model (RM) is developed. The RM is used to determine an optimized system architecture and topology based on a partitioning of functional model components with regard to a set of optimization targets and given system architecture provisions. At the beginning of RM development, aircraft design guidelines, non-functional requirements and constraints for aircraft subsystems, available system architecture components and behavioral model components are determined. These requirements and constraints cover available installation spaces, available resources, characteristics of hardware modules and functions, power provisioning as well as available networks.

After that, the RM is complemented by an optimization model which is used for multiple objective performance optimization of the SuO. In this example, the optimization objective was to minimize overall system cabling in order to decrease overall weight and cost. With this, a Pareto-optimal system architecture can be determined at overall system level. During RM simulation, an allocation of FM components to AM components (partitioning) is determined and stored in the form of an Extensible Markup Language (XML) file. An external generator program, in this example a prototype that was developed by Rath [282] and [30], uses this file in combination with existing BM and AM components to create an executable AM.

Finally, after successful model generation, the created AM is simulated in order to evaluate different design criteria for the SuO. In cases changes are required for any of the three different models, the overall process needs to be repeated until a satisfactory solution has been found. Figure 5.32 depicts the process from BM and RM development to AM generation.

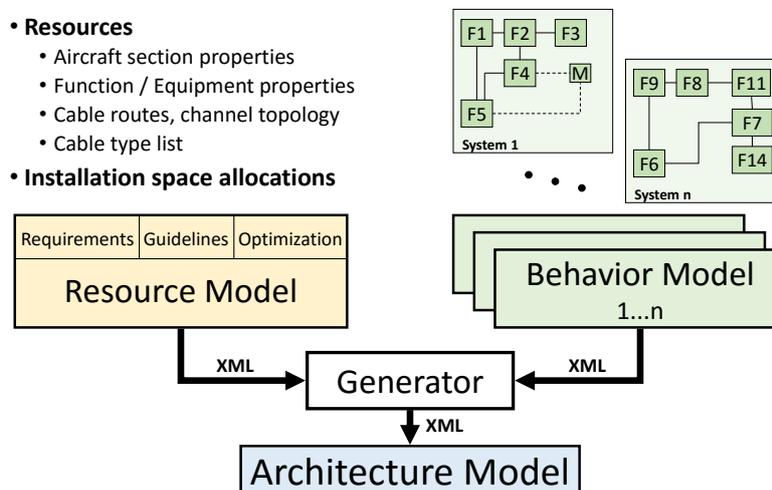


Figure 5.32 – Process for behavior model and resource model development with final architecture model generation (adapted from [124] and [316])

By using executable workflows or simulation sets, it is possible to unite all three phases of system optimization depicted in Figure 5.32 for the given example. In order to do so, existing models for DIMA architecture optimization are used to determine an executable system optimization and validation flow as depicted in Figure 5.33. Based on the original optimization approach, the overall optimization flow can be divided into three top-level simulation steps. Firstly, an optimization model is simulated that was derived from the original RM. Secondly, an external

model generation process is executed and thirdly, the created AM is executed and evaluated with regard to the BM.

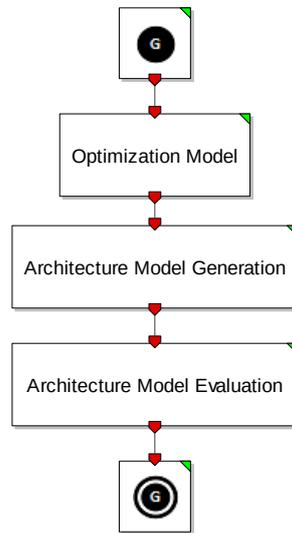


Figure 5.33 – Avionics optimization flow (top-level) in MLDesigner

Figure 5.34 depicts the optimization model that has been created in the form of a simulation set. The original RM was divided into three consecutively executed models.

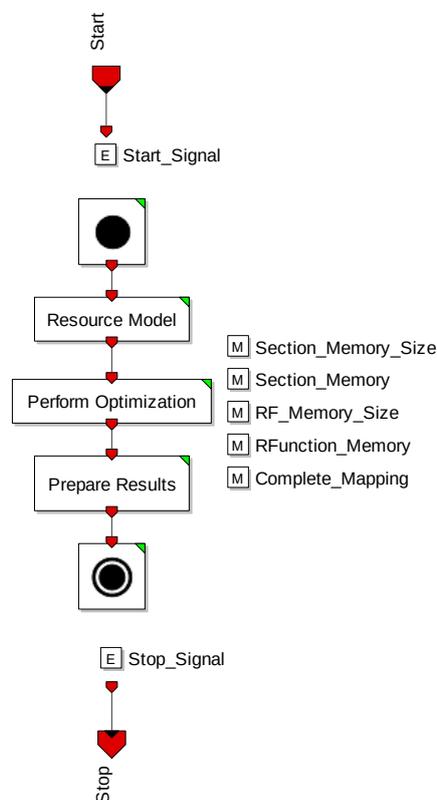


Figure 5.34 – Avionics optimization model in MLDesigner

At the beginning of the optimization process depicted in Figure 5.34, a modified resource model is used to determine non-functional requirements and constraints for the creation of an AM. All information determined by this model is stored within different shared memory modules. During step two, the actual optimization model is executed, based on the information provided by the modified resource model. During the last step, information about the optimization process is compiled and an XML partitioning file is generated.

As part of the next step (cf. Figure 5.33), this file is used by the external generator to create an AM. However, the program developed by Rath [282] and [30] can currently no longer be used for model generation with MLDesigner. This is because the program was a prototype that was specifically tailored for the use within a *opensuse* 10.2 / 10.3 Linux operating environment (released in 2006 / 2007) in combination with a custom model execution target for MLDesigner version 2.7 that was developed by Baumann [30]. As a result, this step has been performed manually in the context of this example. However, as part of the current research project "Automatisierung der Architekturoptimierung komplexer Systeme" (ARKOSE, Engl. automation of the architecture optimization of complex systems, 2014-2017), sponsored by the *German Federal Ministry of Education and Research* under grant number 01IS13031A, a comprehensive architecture model generator will be developed for MLDesigner that can be used to automate this step of optimization flow execution. First results of ARKOSE have already been published, e.g. in reference [368].

Figure 5.35 depicts the internal structure of the AM evaluation model, required for the final step of the architecture optimization flow depicted in Figure 5.33. It is composed of an evaluation report module that is executed in parallel with the original behavior model and the generated architecture model. During simulation, the evaluation report module collects data from the other two models which execute the same mission model. At the end of simulation, different evaluation reports are generated and shown.

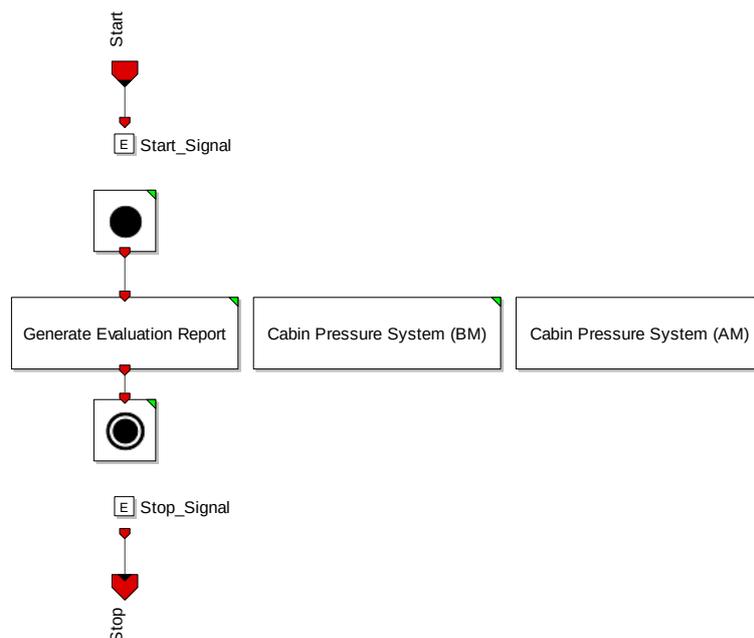


Figure 5.35 – Avionics system architecture evaluation model in MLDesigner

Similar to the example provided in reference [124], a set of values has been determined for sections and functions. In this example, six sections were used with 24 functions for a CPCS as part of a DIMA system architecture. The overall optimization flow was executed with two different settings for optimization of overall cabling length. First by using a first fit algorithm and second by using a first fit decreasing algorithm to allocate functions to sections. Figure 5.36 shows widgets with example results of the optimization process with first fit decreasing algorithm during execution of the optimization model shown in Figure 5.34.

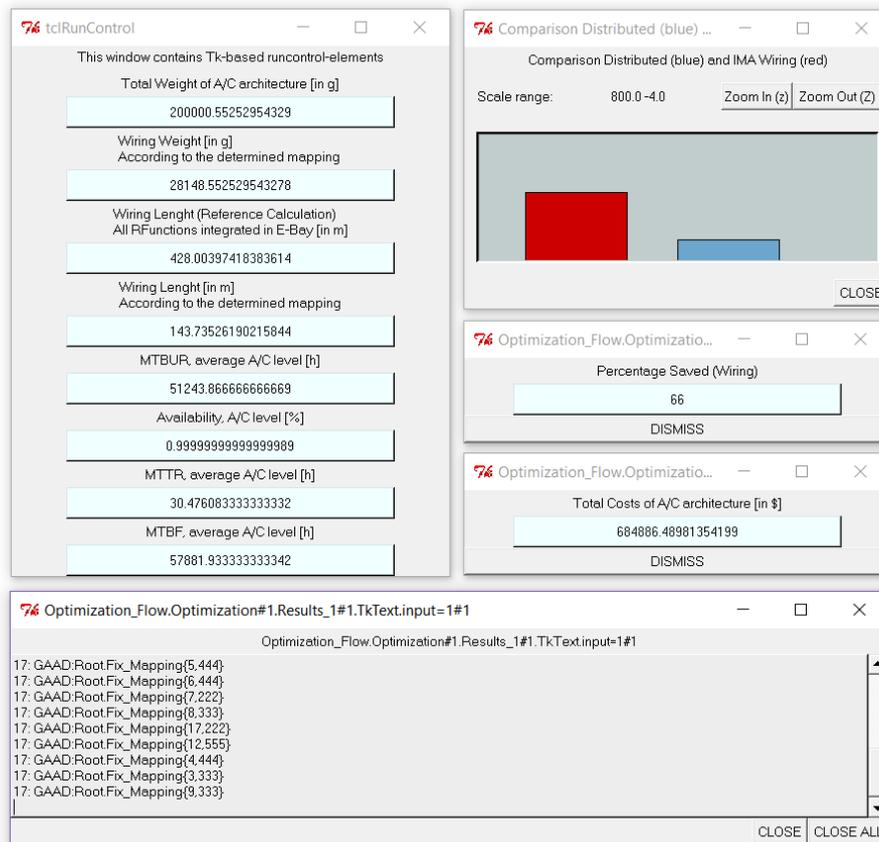


Figure 5.36 – Simulation results in MLDesigner during system architecture optimization with first fit decreasing algorithm

Table 5.5 compares the results of overall system optimization for cabling length based on a first fit algorithm (S1) for function distribution and first fit decreasing algorithm (S2). Values of optimization objective “*Length of Cabling*” are compared to a classical reference system architecture for a CPCS without DIMA, i.e. a federated system architecture with dedicated cabling for different functions that can be found, for instance, in Airbus A320 aircraft.

As shown in Table 5.5, application of both optimization algorithms provides results with significant savings in cabling length. In addition, values for cabling weight and cost decrease for both solutions. With regard to the reference system architecture with 428 m of cabling, using a DIMA architecture with a first fit decreasing algorithm for function distribution, a potential overall saving of 66 % can be observed, resulting in overall cabling length of 143.74 m. Moreover, system architecture S2 has an even higher amount of overall system availability due to differences in function allocations between S1 and S2.

Wert	S1	S2
Saved Cabling (%)	63	66
Length of Cabling [m]	158.52	143.74
Length of Cabling (Reference Architecture) [m]	428.00	428.00
Overall Architecture Weight [g]	214428.74	200000.55
Overall Weight Cabling [g]	42576.74	28148.55
Cost of Architecture [US \$]	1008943.60	684886.49
MTBF [h]	52553.91	57881.93
MTTR [h]	32.37	30.48
MTBUR [h]	46184.89	51243.87
Availability [%]	99.99	>99.99

Table 5.5 – Comparison of two architecture optimization simulations

As part of the last simulation step of the simulation set depicted in Figure 5.34, the generated AM is evaluated together with the FM as shown in Figure 5.35. Both models are simulated with the same mission model, i.e. a flight profile with associated changes in cabin pressure. During this process, values for cabin pressure targets and measured cabin pressure are recorded for comparison. After simulation, a diagram for cabin pressure development is generated as shown in Figure 5.37. This diagram shows cabin pressure target values for BM (blue) and AM (black) together with measured values for BM (red) and AM (green). We observe, that both models provide the same behavior. The difference in timing between both models results from the use of architecture elements, e.g. networks, that delay the execution of CPCS functions or the exchange of information between functions.

By comparing results of the example provided in reference [124] and the results of simulation set execution for a CPCS based on a DIMA system architecture, we observe that both sets of results are consistent. Thus, we may conclude that overall system optimization can successfully be performed by using simulation sets for creating executable optimization flows.

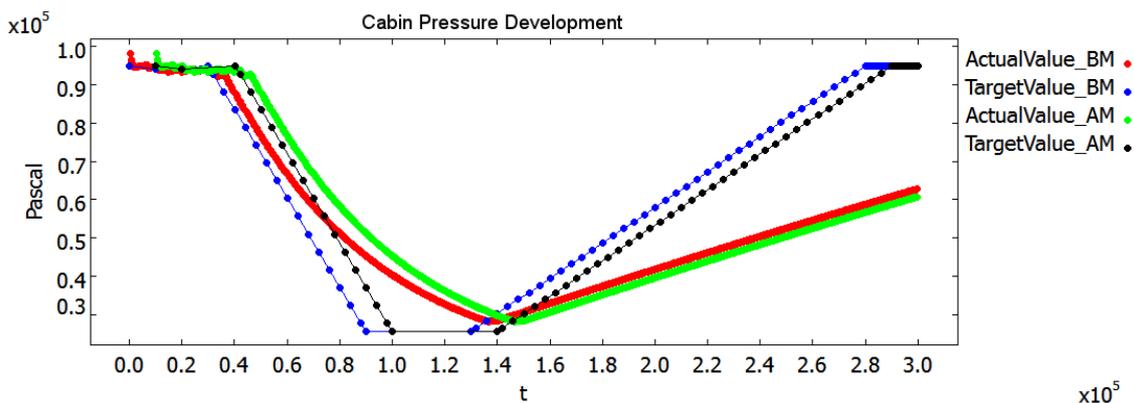


Figure 5.37 – Simulation results for joint evaluation of behavior model and architecture model execution

Annighöfer et al. [17], [18] and [16] independently used Fischer's approach for multi-objective optimization of DIMA architectures and confirmed the optimization potentials of Fischer's approach. The feasibility of system architecture optimization

based on combined system behavior and architecture models and iterative simulation loops has also been demonstrated, e.g. in references [325] and [368]. In another work, Kühn et al. [180] used iterative simulation-based optimization with genetic algorithms for process optimization in hospitals.

5.7 Summary

In this chapter, the application of the developed minimum risk model-based engineering (MR-MBSE) approach described in chapter 4 was demonstrated and validated for the development of cabin management systems (CMS) and avionics for large commercial aircraft. The examples provided in this chapter cover the initial steps of concept development as well as executable requirements specification and executable overall system specification development. The chapter concludes with the elaboration of the results of automated and interactive specification validation and provides an example for the process of automated system architecture optimization.

In the course of CMS development, a subset of 140 major system service requirements has been identified from a set of more than 1000 possible concept requirements. A conceptual design mind map was developed that, at the end of the project, had ten levels of hierarchy and consisted of more than 1200 requirement nodes in total. Based on the conceptual design, a validated executable requirements specification (VERS) was developed and validated successfully in collaboration with CMS stakeholders. During this process, different human machine interface (HMI) and graphical user interface (GUI) design concepts were developed based on current design guidelines for aircraft. Developed conceptual user interface designs were used for virtual prototype (VP) development. By using an MR-MBSE approach, a validated executable overall system specification (VEOSS) has been created successfully for a basic CMS. Moreover, a virtual CMS prototype was created that was integrated in the *Cabin and Cabin Systems Lab* (CCS-Lab) of the *Hamburg University of Applied Sciences*.

Validation activities were performed consistently for each step of development, leading to a system specification with minimized design uncertainty. Necessary changes in specifications due to automated or interactive validation activities were performed as part of an iterative design loop. During automated validation, 1823 different scenario configurations were executed and validated as part of the overall mission model. To finish a complete validation run, the used simulation platforms required approximately 2.42 minutes for model compilation and 26.5 seconds to finish the process of automated validation.

At the end of the project, no remaining validation items were found within the VEOSS. In order to verify the quality of the VEOSS and to confirm validation results, an interactive validation was performed with the CMS prototype. However, it is to be expected that a minimal amount of residual and unknown product uncertainty remains due to possible uncertainties introduced during VERS and VEOSS model development (model uncertainty). Moreover, stakeholder requirements may exist, e.g. excitement needs as described by the Kano model [176], that are yet unknown to stakeholders or will evolve later during system application.

Finally, an existing example from avionics architecture optimization for a cabin pressure control system was successfully used to validate the application of executable optimizations flows based on simulation sets as part of MR-MBSE.

6. Concluding Remarks

In recent years, the complexity of electronic systems with software driven functionality has advanced rapidly. Today's large aerospace systems like the Airbus A380 or the Boeing B787 incorporate a vast array of complex subsystems with thousands of coupled electronic controls units. All subsystems are required to cooperate successfully in order to ensure safe and secure aircraft operation. Hence, aerospace systems are complex system-of-systems that have to be designed to cope with rapidly changing architectures, obsolescence, technological progress and operational challenges. Their main advancements today are driven by progress in readily available electronics and the ability to develop superior systems rapidly at minimum time, cost, weight, power consumption and maximum security.

However, there are no experts or groups of specialists who will understand all of the complex interactions of large-scale systems or even system-of-systems. The use of bottom-up development approaches can neither discover nor resolve design flaws or unwanted emergent behavior that result from dynamic coupling between components, subsystems or the dynamic coupling between systems of systems. In the case of products that include dynamic coupling, even the use of extended periods of testing will not be sufficient in order to determine a feasible design solution. To find the best possible solution for a complex system under design (SuD), e.g. for aircraft, a large number of different designs needs to be integrated and validated as part of an iterative process. But finding an optimal solution between sets of different possible designs is an impossible task if done manually. Moreover, by using bottom-up development approaches, complex systems with dynamic coupling cannot be optimized for a set of top-level requirements.

Thus, written, non-executable specifications in industry have an average design uncertainty of 65% to 70%. As a result, the current chance for success without critical errors for large development projects with investments of several million U.S. dollars is less than 6.4%. The analysis of product uncertainty and development cost over the course of current system development projects shows, that while the majority of overall design uncertainty is introduced early during development, most design problems are resolved late during integration and test or even during operating phase, when costs for redesign are a maximum. In many cases, major design problems are

never resolved because cost overruns also lead to the cancellation of projects. The risk for any system development is the expenditure required for development which is at risk and can be expressed as a function of product uncertainty and development cost per month over the course of development.

Because the currently used methodology for system development solves most design errors at late development stages, when expenditures are a maximum, it has a high risk and leads to failed projects or overruns in development time and cost. Moreover, the currently used design method does not include mechanisms for assuring or optimizing top level requirements. The development risk is a minimum, if specifications include as little uncertainty as possible before starting detailed subsystem development and implementation. This can only be achieved if all development decisions are validated early against the services of a product at mission level by all stakeholders, including the final customer. Thus, it is necessary to shift most design and validation efforts to early stages of concept design and system development, at the upper left hand side of the V-model, when costs are at a minimum. This includes the possibility for automated design iterations and validation in order to be able to determine an optimal design solution at overall system level.

In order to be able to determine an optimized and validated solution for complex development projects, a minimum risk model-based systems engineering (MR-MBSE) methodology with associated design and validation environment was developed, based on the analysis and evaluation of existing solution approaches in industry and science. The application of the developed MR-MBSE method was elaborated from concept design to system specification development, automated validation, virtual prototype development and system design optimization. The overall MR-MBSE process also includes design guidelines for the development of data and customization models. As part of an MR-MBSE development process, the top-down oriented development pyramid is complemented by an inverted validation pyramid. This means, that executable specifications and virtual prototypes, which are created early during development, are validated against the requirements of system services at mission level. Validated executable specifications are used and updated for all decisions from concept development through life cycle management.

To be able to develop executable validation and optimization flows, especially for performing iterative, automated and repeatable steps during a phase-oriented development process, executable workflows and simulation sets have been developed together with associated model components. To support the development of executable specification models, plug-and-play capable model components were developed for mission model, service model and scenario model development. Moreover, model components for validation report generation, automated as well as user-driven interactive validation were developed. All model components developed in this work form an integrated MR-MBSE design and simulation environment that is based on the modeling and simulation tool MLDesigner

Figure 6.1 summarizes the major steps of MR-MBSE for aircraft from concept design to system specification development together with necessary model-based systems engineering (MBSE) skill levels and descriptive keywords (left) in relation to different models, possible tools (middle), and responsible actors (right).

At the start of system development with MR-MBSE, a mission and use case driven concept is developed and validated to determine stakeholder needs. This concept

is used for the creation of an executable requirements specification (ERS). An ERS unites mission requirements, system context, use cases as well as services for the system under design, including functional and performance needs. In parallel, a human machine interface (HMI) concept model is developed. Both models are validated against the conceptual design in order to secure, that all top-level requirements have been determined correctly with regard to stakeholder needs.

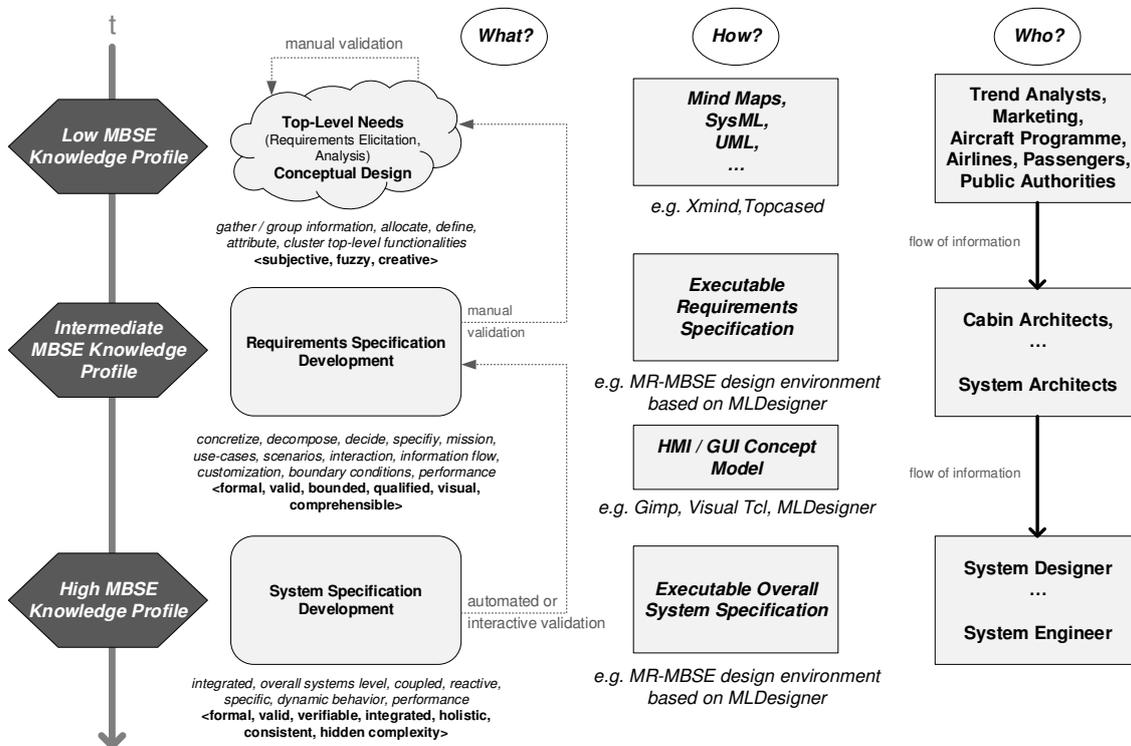


Figure 6.1 – Early steps during minimum risk model-based systems engineering for aircraft with necessary skill levels and descriptive keywords (left) in relation to models, tools (middle), and responsible actors (right) (adapted from Fischer and Wiegmann 2013 [130])

Based on the validated executable requirements specification (VERS), an executable overall system specification (EOSS) is developed. Since system functions, interfaces and architecture are strongly coupled and influence each other, it is important to unite all three aspects within a common executable system specification model. Firstly, in order to be able to validate coupled functional and non-functional requirements. Secondly, in order to evaluate the impact of architecture design decisions on overall system performance and thirdly, to be able to perform architecture optimization before detailed development has started. VERS and EOSS are united to form an executable validation flow (EVF). During simulation of an EVF, the EOSS is automatically validated against the VERS. A validation report at the end of this process shows if the overall validation process was successful. In the case of unsuccessful validation, the report is used to perform necessary re-design steps with subsequent EVF simulation.

The EOSS model is also used in conjunction with the HMI concept model to determine a virtual prototype (VP). As a complement for the automated validation process, this VP can be used for interactive specification validation. This is done by means of user-driven system simulation, i.e. the VP is operated and evaluated

dynamically by stakeholder representatives. Finally, the validated executable overall system specification (VEOSS) is used to perform overall system optimization to determine a product that represents the best possible solution for all stakeholders. This is done by application of executable optimization flows, that unite optimization model, system model generation, automated validation, and performance evaluation. After the successful validation and evaluation of the overall design, the overall system is partitioned into subsystems for detailed development and implementation.

In order to validate and evaluate the developed methodology, examples for the development of cabin management systems (CMS) and avionics for large commercial aircraft were used. It was demonstrated how early validation of development decisions against the services of product at mission level can successfully be achieved with the aid of all stakeholders. The process of CMS development was carried out from early concept design to system specification development, including the development of validated EOSS and VP models. As a result, a validated executable CMS system specification was developed that includes seven major services that are regarded essential for the operation of commercial aircraft. The process of design optimization automation was demonstrated for the development of distributed integrated modular avionics (DIMA) architectures with minimized cabling length for a cabin pressure control system (CPCS). It was shown how simulation sets can be used to determine executable optimization flows that combine optimization model execution, architecture model generation and architecture model evaluation. As a result of the simulation of the developed optimization flow, potential savings of up to 66% for overall cabling were determined.

This work shows how modeling and simulation at mission level permits the creation of early conceptual models and executable specifications that can be used for automated validation, the creation of virtual prototypes and early design optimization. By using an MR-MBSE approach, most efforts for system design, redesign and validation are shifted to early development stages of the upper left hand side of the V-model, when development costs are a minimum. The application of automated and interactive validation early during development leads to quick design iteration cycles, especially in terms of stakeholder-oriented specification validation and the possibility of early testing. As a result, product uncertainty is reduced much earlier than during traditional development approaches as depicted in Figure 6.2.

In doing so, the probability of critical design errors is decreased as early as possible, thus minimizing the overall risk of development. Salzwedel et al. [317] determined a ratio of risk of 4.9 between traditional and mission level-based development processes, which constitutes a significant risk reduction. Using executable specifications from concept design to specification development also increases the amount of re-use for following development projects with similar scope. In addition, the use of formal executable overall system specifications allows to perform formal verification and model-checking activities early during development. The developed approach also contributes to solving the complexity challenge in avionics and cabin systems design. Moreover, the development of simulation sets contributes to the development of future design and process automation strategies.

Currently, fully automated architecture model generation is not possible with MLDesigner, due to changes in the overall software architecture. Moreover, it is currently not possible to use simulation sets for distributed model execution for a set of remote

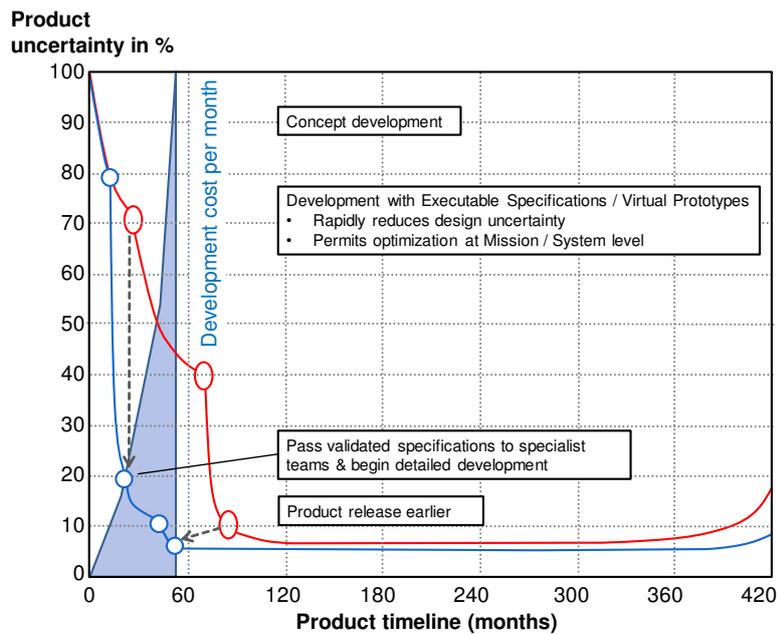


Figure 6.2 – Early minimization of product uncertainty when development costs are still low to minimize the risk of development by application of a minimum risk model-based systems engineering approach (adapted from Salzwedel et al. 2008, 2009, 2010 [316], [317] and [215])

computers, e.g. a computer cluster. A solution for combined architecture model generation, optimization and distributed simulation is currently under development as part of a research project called "Automatisierung der Architekturoptimierung komplexer Systeme" (ARKOSE, Engl. automated architecture optimization for complex systems, 2014-2017).

It has also become evident, that the application of an MR-MBSE approach introduces new development challenges compared to traditional requirements engineering approaches. With MR-MBSE, in order to make specification models executable, it is crucial for system architects and engineers to make concrete design decisions for mission and service models early during development. This may be criticized as an early anticipation of a possible implementation. However, system engineers and programmers of later development stages are required to find solutions to specification requirements, but cannot evaluate the impact of their decisions on the overall system. In case of vague system requirements, implementers may provide solutions in a way that is not optimal for the overall system. Thus, by providing concrete specifications and not omitting or postponing major design decisions to late development stages, the overall quality of specifications is significantly increased.

Currently, no method has been developed in the context of MR-MBSE for the phase of implementation based on executable specification models. In the case of software development, automated source code generation might offer a solution for this transition. From a system certification perspective, especially for aircraft, it is unclear if and how executable specifications can be used in collaboration with public authorities in order to provide an adequate level of confidence. Thus, it is necessary to discuss the future process of certification with public as well as other certification-related authorities. The same applies for the process of forming legally binding contracts between different stakeholders, e.g. between system manufacturer and subsystem suppliers. The process of distributed working as part of a large devel-

opment teams should also be discussed as part of future MR-MBSE improvements as well as the integration of the developed design and validation environment into existing tool chains.

Although functional requirements can be completely integrated within executable specifications, a set of non-functional requirements may exist that are hard to include, especially for the process of automated validation. These requirements include physical design properties including colors, shapes, form factors or dimensions, building or installation specific requirements as well as properties that result from subjective impressions, e.g. noises created by a final design solution. Currently, these kinds of requirements can only be integrated in the form of model annotations. Since non-functional requirements of executable specifications need to be quantifiable, it may sometimes be hard for system engineers to take a decision early during development. In such cases, domain-specific knowledge, work experience and engineering judgment may be an advantage. Moreover, the use of discrete event models in MLDesigner cannot currently provide real parallelism during simulation set or mission model execution. So far, it is only possible to perform quasi parallel mission and service model execution.

The development of atomic mission models (use cases) may lead to a large number of actor modules that need to be integrated in the EOSS model. In order to keep EOSS model comprehensibility at a high level, it is advisable to aggregate similar actor modules or to reduce the overall number of different actors. In the case of service and scenario models, the risk of scenario explosion exists. Thus, it is advisable to determine only essential services for each use case. Moreover, scenario models have a potential risk of creating deadlocks or infinite loops during overall system simulation. This is currently remedied by using time-out mechanisms and thorough model reviews. In future extensions for executable workflows and simulation sets, this should be improved, e.g. by using extended model checking. When creating scenario models with automated validation for large sets of customization model parameter combinations (exponential design space), e.g. by using brute force combination approaches, it is possible to create simulations that need an enormous amount of time to execute. In order to avoid this, it may be necessary to use heuristics or stochastic approaches to determine a feasible solution.

Other possible extensions for the developed MR-MBSE method exist that should be investigated as part of future research. In order to cover more parts of the overall product life cycle, the range of MR-MBSE should be extended to include later development stages, e.g. implementation and test. Thus, an automated test process with executable specifications should be developed. Moreover, requirements management tools need to be linked to executable specifications in order to keep track of the overall development process and model changes. Finally, the application of MR-MBSE needs to be analyzed and evaluated as part of other complex system development projects in order to be able to determine domain specific extensions for the developed methodology.

Bibliography

- [1] ESSI Project No. 11000. *ESPITI Study: European Software Process Improvement*. European Software Process Improvement Training Initiative, European Commission, 1995 / 1996.
- [2] Federal Aviation Administration. FAA Aerospace Forecast Fiscal Years 2013-2033. http://www.aia-aerospace.org/assets/FAA_2013_to_2033_Aerospace_Forecast.pdf, 2013. USA, available online, retrieved 07/14/2014.
- [3] Ankur Agarwal, Cyril-Daniel Iskander, Ravi Shankar, and Georgiana Hamza-Lup. System-Level Modeling Environment: MLDesigner. In *2nd Annual IEEE Systems Conference*, pages 1–7, April 2008.
- [4] European Aviation Safety Agency. EU-OPS 1, Commission Regulation (EC) No859/2008. Official Journal of the European Union, L254, 20 August 2008, amending Council Regulation (EEC) No 3922/91 as regards common technical requirements and administrative procedures applicable to commercial transportation by aeroplane, 2008.
- [5] Smithsonian Air and Space Museum. Z1 1936: Static Aircraft Stress due to Distributed Aerodynamic Loads, Z3 1941: Flutter Analysis. Museum Exhibition, Washington, D.C., USA, 2011.
- [6] Boeing Commercial Airplanes. Boeing Current Market Outlook 2015-2034. http://www.boeing.com/resources/boeingdotcom/commercial/about-our-market/assets/downloads/Boeing_Current_Market_Outlook_2015.pdf, 2015. USA, available online, retrieved 22/10/2016.
- [7] Yoji Akao. *Quality Function Deployment (QFD): Integrating Customer Requirements into Product Design*. Productivity Press, pages 3-25, 11 2004.
- [8] Mack W. Alford. A Requirements Engineering Methodology for Real-Time Processing Requirements. *Software Engineering, IEEE Transactions on*, SE-3(1):60–69, Jan 1977.
- [9] Mack W. Alford. Software Requirements Engineering Methodology (SREM) at the age of two. In *The IEEE Computer Society's Second International Computer Software and Applications Conference (COMPSAC '78)*, pages 332–339, 1978.
- [10] Mack W. Alford. SREM at the Age of Eight; The Distributed Computing Design System. *Computer*, 18(4):36–46, April 1985.

- [11] Stewart Allen. Visual Tcl. <http://vtcl.sourceforge.net/>, 2010. open source Tcl/Tk application development environment, available online, retrieved 07/15/2014.
- [12] Hans-Henrich Altfeld. *Commercial Aircraft Projects: Managing the Development of Highly Complex Products*. Ashgate Publishing Limited, pages 6 et seq., 13 et seq., 18 et seq., 27 et seq., 45, 79, 103, 2010.
- [13] Mahmood Ameen and Vijay Rathnam. Performance monitoring of IFE systems - Proposal for the future from Emirates Airline perspective. *FAST 38 - Flight Airworthiness Support Technology*, 38:2–9, July 2006. available online, retrieved 07/11/2014.
- [14] Peder A. Andersen. New Civil Aircraft Competitors on the Horizon? *U.S. International Trade Commission, DOI: 10.2139/ssrn.1445029*, April 2009.
- [15] Derek E. Anderson and Mark W. Beranek. 777 Optical LAN Technology Review. *48th IEEE Electronic Components and Technology Conference*, pages 386–390, 25 May - 28 May 1998.
- [16] Björn Annighöfer. *Model-based Architecting and Optimization of Distributed Integrated Modular Avionics (Schriftenreihe Flugzeug-Systemtechnik)*. Shaker, 1 edition, 2 2015.
- [17] Björn Annighöfer, Ernst Kleemann, and Frank Thielecke. Model-based Development of Integrated Modular Avionics Architectures on Aircraft-level. *Deutscher Luft- und Raumfahrtkongress (DLRK 2011)*, 27 - 29 September 2011.
- [18] Björn Annighöfer and Frank Thielecke. Multi-objective mapping optimization for distributed Integrated Modular Avionics. In *IEEE/AIAA 31st Digital Avionics Systems Conference (DASC)*, pages 6B2–1–6B2–13, October 2012.
- [19] Pedro Argüelles, Manfred Bischoff, Philippe Busquin, B.A.C. Droste, Sir Richard Evans, Walter Kröll, Jean-Luc Lagardère, Alberto Lina, John Lumsden, Denis Ranque, Søren Rasmussen, Paul Reutlinger, Sir Ralph Robins, Helena Terho, and Arne Wittlöv. *European Aeronautics: a Vision for 2020, Meeting Society's Needs and Winning Global Leadership*. ISBN 92-894-0559-7, JANUARY 2001. Published by the European Commission, Advisory Council for Aviation Research and Innovation in Europe (ACARE).
- [20] Bradford Henry Arnold. *Logic and Boolean Algebra (Dover Books on Mathematics)*. Dover Publications, Inc., New York, reprint edition, 10 2011.
- [21] Dave Arnold, Jean-Pierre Corriveau, and Wei Shi. Scenario-Based Validation: Beyond the User Requirements Notation. In *21st Australian Software Engineering Conference (ASWEC 2010)*, pages 75–84, April 2010.
- [22] James D. Arthur, Markus K. Gröner, Kelly J. Hayhurst, and C. Michael Holloway. Evaluating the Effectiveness of Independent Verification and Validation. *Computer*, 32(10):79–83, Oct 1999.
- [23] Dirk Asendorpf. Das große Flattern. *Die Zeit*, 1:35, December 2004.

- [24] Peter Ashenden. *The Designer's Guide to VHDL, 3rd revised edition*. Morgan Kaufmann, 2006.
- [25] N. R. Augustine. Today... tomorrow [of multidisciplinary systems of systems]. *IEEE Aerospace and Electronics Systems Magazine, Jubilee Issue Vol 15. No 10.*, pages 137–144, February 2000.
- [26] Mike Badalament. *Concepts and Trends in Virtual ECU Software Development*. ETAS Inc., SAE International, SAE Vehicle Engineering Webcast, available to audience members until 05/09/2013, retrieved 05/09/2012. <http://www.sae.org/mags/sve/webcasts.htm>.
- [27] Jerry Banks, John Carson, Barry L. Nelson, and David Nicol. *Discrete Event System Simulation*. Pearson, 2005.
- [28] Ricardo M. Bastos and Duncan Dubugras A. Ruiz. Extending UML activity diagram for workflow modeling in production systems. In *Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS)*, pages 3786–3795, Jan 2002.
- [29] Tommy Baumann. *Integrierte Hard- und Softwareentwicklung mit dem MLDesigner Entwicklung einer UML-Schnittstelle*. Faculty Computer Science & Automation, University of Technology Ilmenau, July 2004. Master Thesis (Diplomarbeit).
- [30] Tommy Baumann. *Automatisierung der frühen Entwurfsphasen verteilter Systeme: Eine Methode zur frühzeitigen Analyse und Optimierung von Gesamtsystemarchitekturen zur Steigerung von Spezifikationsqualität und -geschwindigkeit*. Südwestdeutscher Verlag für Hochschulschriften, 2009.
- [31] Anthony Beachy. Cabin Fever? *Aerospace Technology*, 23 January 2008. available online, retrieved 12/07/2011.
- [32] Peter Bearman, John Green, Ros Howell, Jeff Jupp, Simon Luxmoore, Geoff Maynard, Kevin Morris, Ian Poll, Hugh Somerville, Robert Whitfield, and Roger Wiltshire. Air Travel - Greener by Design, Annual Report 2012-2013. <http://aerosociety.com/Assets/Docs/Greener%20by%20Design/GbD%20report%202013%20v3.pdf>, 2013. Royal Aeronautical Society, United Kingdom, issued annually, available online, retrieved 07/15/2014.
- [33] Ottmar Bender. Erkenntnisse aus der Entwicklung komplexer und sicherheitskritischer Avionik-Architekturen. In *Object Oriented Programming Conference (OOP 2011)*, Munich, Germany, January 24-28 2011.
- [34] Jeff Bergenthal. Final Report Model Based Engineering (MBE) Subcommittee. *Report: NDIA Systems Engineering Division M&S Committee*, April 2011.
- [35] Jeff Bergenthal. Final Report Model Based Engineering (MBE) Subcommittee - National Defense Industrial Association (NDIA) Systems Engineering Division M&S Committee. *14th Annual Systems Engineering Conference*, 24 / 27 October 2011.

- [36] Sven Olaf Berkhahn. *Lecture on electronic cabin systems for aircraft (Elektronische Kabinensysteme, Teil 2)*. Hamburg University of Applied Sciences (HAW-Hamburg), 2009.
- [37] Philip J. Birtles. *Boeing 777: Jetliner for a New Century (Airliner Color History)*. Motorbooks International, 1998.
- [38] Randall H. Black. Rapid development of avionics systems. *The AIAA/IEEE/SAE Digital Avionics Systems Conference - 17th DASC*, pages H14-1 – H14-6, 31 October - 7 November 1998.
- [39] Benjamin S. Blanchard and Wolter J. Fabrycky. *Systems Engineering and Analysis*. Pearson Education Limited, 5th edition, 7 2014.
- [40] Mars Climate Orbiter Mishap Investigation Board. Mars Climate Orbiter Mishap, Phase I Report. *National Aeronautics and Space Administration (NASA)*, press release, November 10 1999.
- [41] Thomas Bochot, Pierre Virelizier, H el ene Waeselynck, and Virginie Wiels. Model Checking Flight Control Systems: the Airbus Experience. *31st International Conference on Software Engineering - Companion Volume (ICSE-Companion 2009)*, pages 18–27, 16-24 May 2009.
- [42] Thomas Mathias Bock. Changes in methodologies for cabin interior design: From drawings to virtual reality simulations. personal communication, interview, 10 September 2013. Senior Cabin / Interior Designer at Airbus (1974 - 2009).
- [43] Barry Boehm and Jo Ann Lane. Using the Incremental Commitment Model to Integrate System Acquisition, Systems Engineering, and Software Engineering. *University of Southern California (USC), an expanded version of CrossTalk (Foundation for Cross-Connection Control and Hydraulic Research), USC-CSSE-2007-715*, October 2007.
- [44] Barry W. Boehm. Software Engineering. *IEEE Transactions on Computers*, C-25(12), December 1976.
- [45] Barry W. Boehm. Guidelines for Verifying and Validating Software Requirements and Design Specifications. *European Conference on Applied Information Technology of the International Federation for Information Processing (EURO IFIP 79)*, pages 711–719, 25-28 September 1979. London, UK, North-Holland Publishing Company, Paul A. Samet (ed.).
- [46] Barry W. Boehm. *Software Engineering Economics*. Prentice-Hall, Advances in Computing Science & Technology Series, 1981.
- [47] Barry W. Boehm. Verifying and Validating Software Requirements and Design Specifications. *IEEE Software*, 1(1), January 1984.
- [48] Barry W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61 – 72, 1988.

- [49] Andreas Borg, Angela Yong, Pär Carlshamre, and Kristian Sandahl. The Bad Conscience of Requirements Engineering: An Investigation in Real-World Treatment of Non-Functional Requirements. *Proceedings of the Third Conference on Software Engineering Research and Practice in Sweden*, pages 1–8, 2003.
- [50] Jürgen Bortolazzi. *Untersuchung zur rechnergestützten Erfassung, Verwaltung und Prüfung von Anforderungsspezifikationen und Einsatzbedingungen elektronischer Steuerungs- und Regelungssysteme*. PhD thesis, Technische Universität Erlangen-Nürnberg, 1994. Ph. D. Dissertation.
- [51] Jonathan P. Bowen and Michael G. Hinchey. Seven More Myths of Formal Methods. *IEEE Software*, 12(4):34–41, July 1995.
- [52] Marco Bozzano and Adolfo Villaflorida. *Design and Safety Assessment of Critical Systems*. CRC Press, Auerbach Publications, Taylor & Francis Group LLC, 2010.
- [53] S. Bradner. Key words for use in RFCs to Indicate Requirement Levels. RFC 2119 (Best Current Practice), March 1997. retrieved 12/19/2014.
- [54] Deborah M. Braid, Cris W. Johnson, and Glenn D. Schillinger. An integrated test approach for avionics system development. *20th Digital Avionics Systems Conference (DASC 2001)*, pages 9B2/1 – 9B2/9 vol.2, October 2001.
- [55] Air Accidents Investigation Branch. AAIB Bulletin S2/2013 SPECIAL. http://www.aaib.gov.uk/cms_resources/S2-2013%20G-REDW%20G-CHCN.pdf, 2013. AAIB investigation report EC225 Helicopter accidents, Report name: S2/2013 EC225 LP, G-REDW and G-CHCN, Published 18 March 2013, retrieved 10/10/2014.
- [56] Dominique Brière and Pascal Traverse. AIRBUS A320/A330/A340 Electrical Flight Controls A Family of Fault-Tolerant Systems. *The Twenty-Third International Symposium on Fault-Tolerant Computing - FTCS-23*, pages 616–623, 22-24 June 1993.
- [57] Eckard Bringmann and Andreas Krämer. Model-Based Testing of Automotive Systems. In *1st International Conference on Software Testing, Verification, and Validation*, pages 485–493, Lillehammer, Norway, April 2008.
- [58] Encyclopædia Britannica. "complexity", subsection emergent behaviour. <http://www.britannica.com/EBchecked/topic/130050/complexity/129410/Emergent-behaviour>, 2012. Encyclopædia Britannica Online, Encyclopædia Britannica Inc., retrieved 12/06/2012.
- [59] Encyclopædia Britannica. "avionics". <http://www.britannica.com/EBchecked/topic/45839/avionics>, 2013. Encyclopædia Britannica Online, Encyclopædia Britannica Inc., retrieved 02/08/2013.
- [60] Encyclopædia Britannica. "empiricism". <http://www.britannica.com/EBchecked/topic/186146/empiricism>, 2013. Encyclopædia Britannica Online, Encyclopædia Britannica Inc., retrieved 01/08/2013.

- [61] Encyclopædia Britannica. "rationalism". <http://www.britannica.com/EBchecked/topic/492034/rationalism>, 2013. Encyclopædia Britannica Online, Encyclopædia Britannica Inc., retrieved 01/08/2013.
- [62] Manfred Broy, Sascha Kirstan, Helmut Kremer, Bernhard Schätz, and Jens Zimmermann. *What is the benefit of a model-based design of embedded software systems in the car industry?* Emerging Technologies for the Evolution and Maintenance of Software Models, pages 343-369, 2011.
- [63] Manfred Broy and Oscar Slotosch. Enriching the Software Development Process by Formal Methods. *Applied Formal Methods - FM-Trends 98, International Workshop on Current Trends in Applied Formal Method*, pages 44–61, October 7-9 1998.
- [64] Dennis M. Buede. *The Engineering Design of Systems: Models and Methods*. John Wiley & Sons, pages 6 et seq., 46 et seq., 2000.
- [65] Stefan Burger, Oliver Hummel, and Matthias Heinisch. Airbus Cabin Software. *IEEE Software*, 30(1):21–25, January / February 2013.
- [66] Ed Burnette and Jörg Staudemeyer. *Eclipse IDE - kurz & gut*. O'Reilly, 2009.
- [67] Robert K. Butler, Lawrence C. Creech, and Albert J. Anderson. System Architecture and Operational Concept Validation through Modeling and Simulation. *IEEE Military Communications Conference (MILCOM 2006)*, pages 1–6, 22 et seq., 23-25 October 2006.
- [68] Henning Butz. Open Integrated Modular Avionic (IMA): State of the Art and future Development Road Map at Airbus Deutschland. <http://www.aviation-conferences.com/architecture/pdf/ima-henning-butz.pdf>, 2008. Aircraft Systems Technologies Workshop, Hamburg, Department of Avionic Systems at Airbus Deutschland GmbH, available online, retrieved 02/08/2013.
- [69] Henning Butz. Nachhaltiger Nutzen formaler Methoden bei der Entwicklung und Integration aero-nautischer Produkte / How to Gain Sustainable Value from the Application of Formal Methods in the Development and Integration of Aero-Nautic Products. In *Innovation Forum Embedded Systems, Bavarian Information and Communication Technology Cluster (BICC^{NET})*, Munich, Germany, April 23 2010.
- [70] Tony Buzan and Barry Buzan. *The Mind Map Book*. BBC Active, 12 2009.
- [71] Luis Cabral and Tobias Kretschmer. Competition in the Wide-Body Aircraft Market. *New York University (NYU) Leonard N. Stern School of Business, lecture note*, August 2001.
- [72] George Cancro, Russell Turner, Eli Kahn, and Steve Williams. Executable Specification-Based System Engineering. *IEEE Aerospace Conference 2011*, pages 1–9, 5-12 March 2011.
- [73] Bill Carey. Cabin Comfort On The A380. *Avionics Magazine - Solutions for Global Airspace Electronics*, January 2008. available online, retrieved 07/23/2014.

- [74] Paul G. Carlock and Robert E. Fenton. System of Systems (SoS) enterprise systems engineering for information-intensive organizations. *Systems Engineering*, 4(4):242–261, 2001.
- [75] John M. Carroll. Introduction: The Scenario Perspective on System Development. In *Scenario-Based Design: Envisioning Work and Technology in System Development*. John Wiley & Sons Inc, 5 1995.
- [76] John M. Carroll. Five Reasons for Scenario-based Design. *Interacting with Computers*, 13(1):43–60, 2000.
- [77] Yolanda Carson and Anu Maria. Simulation Optimization: Methods And Applications. In *Proceedings of the 1997 Winter Simulation Conference*, pages 118–126, Dec 1997.
- [78] Andrew Chaikin. Is SpaceX Changing the Rocket Equation? 1 visionary + 3 launchers + 1,500 employees = ? *Air & Space/Smithsonian*, JANUARY 2012, 2012.
- [79] Lawrence Chung and Julio Cesar Sampaio do Prado Leite. On Non-Functional Requirements in Software Engineering. *Lecture Notes in Computer Science (LNCS)*, Springer-Verlag Berlin Heidelberg, 5600:363–379, 2009.
- [80] Russell W. Claus, Thomas Lavelle, Scott Townsend, and Mark Turner. Coupled High-fidelity Engine System Simulation. *26th International Congress of the Aeronautical Sciences (ICAS 2008)*, 14 - 19 September 2008.
- [81] John R. Clymer. *Simulation-Based Engineering of Complex Systems*. John Wiley & Sons, Inc., pages xxi, 1 et seq., 31 et seq., 41 et seq., 55 et seq., 2 edition, 3 2009.
- [82] Alistair Cockburn. Structuring Use Cases with Goals. *Journal of Object-Oriented Programming*, published in two parts, September/October and November/December 1997.
- [83] Mike Cohn. *User Stories Applied: For Agile Software Development (Addison-Wesley Signature)*. Addison Wesley, 1 edition, 2004.
- [84] Mike Cohn. *Succeeding with Agile: Software Development Using Scrum*. Addison-Wesley Longman, Amsterdam, 2009.
- [85] Airlines Electronic Engineering Committee. *ARINC Specification 664P7 - Aircraft Data Network Part 7 Avionics Full Duplex Switched Ethernet (AFDX) Network*. Aeronautical Radio, Incorporated (ARINC), 2005.
- [86] Airlines Electronic Engineering Committee. *ARINC 837 - Design Guidelines for Aircraft Cabin Human Machine Interfaces*. Aeronautical Radio, Incorporated (ARINC), 2011.
- [87] The Boeing Company. Boeing 747 Family Fun Facts. http://www.boeing.com/commercial/747family/pf/pf_facts.html, 1995. retrieved 11/01/2011.

- [88] Larry L. Constantine and Lucy A. D. Lockwood. *Software for Use: A Practical Guide to the Methods of Usage-Centered Design: A Practical Guide to the Models and Methods of Usage-centered Design*. Addison Wesley Pub Co Inc, 5 1999.
- [89] Alan Cooper. *About Face: Essentials of Window Interface Design*. Wiley Publishing Inc., 3rd ed., 2007.
- [90] Kenneth G. Cooper. The Rework Cycle (a series of 3 articles): Why projects are mismanaged; How it really works ...and reworks; Benchmarks for the project manager. *PMNETwork (appeared also in: Project Management Journal, March 1993)*, 7(2):25–28, February 1993.
- [91] Ivica Crnkovic, Michel Chaudron, and Stig Larsson. Component-Based Development Process and Component Lifecycle. In *International Conference on Software Engineering Advances*, pages 44–53, Oct 2006.
- [92] W.A. Crossley. Systems of Systems: An Introduction of Purdue University Schools of Engineerings Signature Area. *Engineering Systems Symposium, Cambridge, MA*, March 31, 2004.
- [93] V. Curcin and M. Ghanem. Scientific workflow systems - can one size fit all? In *Biomedical Engineering Conference, 2008. CIBEC 2008. Cairo International*, pages 1–9, Dec 2008.
- [94] William J. Dally and Charles L. Seitz. The Torus Routing Chip, California Institute of Technology, Computer Science Technical Reports, CaltechCSTR:1986.5208-tr-86, ID 26909, CaltechCSTR Collection. <http://authors.library.caltech.edu/26909/>, 24 January 1986. available online, retrieved 08/01/2012.
- [95] Werner Damm, Sylvain Prudhomme, and Albert Benveniste. Controlling Speculative Design using Rich Component Models. *IRISA Tech Seminar (Séminaires Irisatech), 34ème rencontre Irisatech, Institut national de recherche en informatique et en automatique (INRIA)*, 2005.
- [96] Oliver Döbertin. Simplifizierte Integration (SINTEG): Schlussbericht des Verbundvorhabens ; Luftfahrtforschungsprogramm LUFO IV 2nd Call; Laufzeit des Vorhabens: 01.01.2009 - 30.09.2011. final project report (public), Airbus Operations GmbH; Reportnr. / Förderkennzeichen: 20K0806A, Verbund-Nr. 01067253, 2012. Technische Informationsbibliothek und Universitätsbibliothek Hannover (TIB/UB), DOI: 10.2314/GBV:725535237.
- [97] Oliver Döbertin. Simplifizierte Kabine (SIMKAB): Schlussbericht des Verbundvorhabens ; Luftfahrtforschungsprogramm LUFO IV 2nd Call; Laufzeit des Vorhabens: 01.01.2009 - 30.09.2012. final project report (public), Airbus Operations GmbH; Reportnr. / Förderkennzeichen: 20K0805A, Verbund-Nr. 01066673, 2013. Technische Informationsbibliothek und Universitätsbibliothek Hannover (TIB/UB), DOI: 10.2314/GBV:772483558.
- [98] John D. Burroughs and Ronald A. Hammond. Control Analysis and Design Features of EASY5. *Proceedings of the 1983 American Control Conference*, pages 58–63, 22–24 June 1983.

- [99] J.P. Potocni de Montalk. Computer software in civil aircraft. *Proceedings of the IEEE/AIAA 10th Digital Avionics Systems Conference (DASC)*, pages 324 – 330, 14-17 October 1991.
- [100] Dionisio de Niz, Peter H. Feiler, David P. Gluch, and Lutz Wrage. A Virtual Upgrade Validation Method for Software-Reliant Systems. <http://www.sei.cmu.edu/library/abstracts/reports/12tr005.cfm>, 2012. TECHNICAL REPORT - (CMU/SEI-2012-TR-005), ESC-TR-2012-005, Software Engineering Institute, Carnegie Mellon University.
- [101] Ewa Deelman, Dennis Gannon, Matthew Shields, and Ian Taylor. Workflows and e-Science: An overview of workflow system features and capabilities. *Future Generation Computer Systems*, 25(5):528 – 540, 2009.
- [102] Douglas Densmore, Roberto Passerone, and Alberto Sangiovanni-Vincentelli. A Platform-Based Taxonomy for ESL Design. *IEEE Design Test of Computers*, 23(5):359–374, September-October 2006.
- [103] Fassely Doumbia, Odile Laurent, Chantal Robach, and Michel Delaunay. Using the Testability analysis methodology for the Validation of AIRBUS Systems. *First International Conference on Advances in System Testing and Validation Lifecycle (VALID'09)*, pages 86–91, 20-25 September 2009.
- [104] Marlon Dumas and Arthur H.M. ter Hofstede. UML Activity Diagrams As a Workflow Specification Language. In *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools, UML'01*, pages 76–90, London, UK, UK, 2001. Springer-Verlag.
- [105] Tom Durkin. What the Media Couldn't Tell You About the Mars Pathfinder. *Robot Science & Technology 1 (RobotMag), Issue 1*, 1998.
- [106] European Aviation Safety Agency (EASA). Certification Specifications and Acceptable Means of Compliance for Large Aeroplanes (CS-25), Amendment 15. <http://easa.europa.eu/agency-measures/certification-specifications.php#CS-25>, 2014. retrieved on 10/13/2014.
- [107] Steve Easterbrook. The difference between Verification and Validation. <http://www.easterbrook.ca/steve/?p=2030>, November 29th, 2010. Science Blog, University of Toronto, Toronto, Ontario, Canada, retrieved 11/27/2012.
- [108] Christof Ebert. *Systematisches Requirements Engineering: Anforderungen ermitteln, spezifizieren, analysieren und verwalten*. dpunkt.verlag GmbH, 2012.
- [109] Paul Eden. *Civil Aircraft Today The World's Most Successful Commercial Aircraft*. Amber Books Ltd., 2008.
- [110] Paul Eden. Less is more / Shedding light on LEDs. *Aircraft Cabin Management*, 2:8–13, April 2013.
- [111] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorfer, S. Sachs, and Y. Xiong. Taming Heterogeneity-the Ptolemy Approach. *Proceedings of the IEEE*, 91(1):127 – 144, 2003.

- [112] Jeff A. Estefan. Survey of Model-Based Systems Engineering (MBSE) Methodologies. *International Council on Systems Engineering (INCOSE), INCOSE Technical Publication, Document No.: INCOSE-TD-2007-003-01, Version/Revision: B*, June 10 2008.
- [113] Stephen Uhler et al. SpecTcl and GUI Builder. <http://spectcl.sourceforge.net/>, 2006. open source Tcl/Tk application development environment, available online, retrieved 07/15/2014.
- [114] Peter H. Feiler. Open Source AADL Tool Environment (OSATE). http://www.aadl.info/aadl/documents/d1_1500_Osate.pdf, 2005. AADL Workshop Paris, France, available online, retrieved 01/18/2013.
- [115] Peter H. Feiler. Model-based Validation of Safety-Critical Embedded Systems. In *IEEE Aerospace Conference 2010*, pages 1–10, March 2010.
- [116] Peter H. Feiler. SAE AADL V2: An Overview. *Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA*, 2010.
- [117] Peter H. Feiler. Model-based Engineering of Software-Reliant Systems with AADL. *INCOSE Webinar Series, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA*, May 2010.
- [118] Peter H. Feiler and Dionisio de Niz. ASSIP Study of Real-Time Safety-Critical Embedded Software-Intensive System Engineering Practices (CMU/SEI-2008-SR-001). <http://www.sei.cmu.edu/library/abstracts/reports/08sr001.cfm>, February 2008. Software Engineering Institute, Carnegie Mellon University, Army Strategic Software Improvement Program (ASSIP).
- [119] Peter H. Feiler and David P. Gluch. *Model-based Engineering with AADL: An Introduction to the SAE Architecture Analysis and Design Language*. Addison-Wesley Longman, 2012.
- [120] Peter H. Feiler, Jorgen Hansson, Dionisio de Niz, and Lutz Wrage. System Architecture Virtual Integration: An Industrial Case Study. Technical Report CMU/SEI-2009-TR-017, ESC-TR-2009-017, November 2009. Research, Technology, and System Solutions (RTSS) Program, Aerospace Vehicle System Institute (AVSI), Carnegie Mellon University.
- [121] Wagner Ferdinand, Schmuki Ruedi, Wolstenholme Peter, and Wagner Thomas. *Modeling Software with Finite State Machines: A Practical Approach*. CRC Press, Auerbach Publications, Taylor & Francis Group, pages 11 et seq., 13 et seq., 47 et seq., 49 et seq., 2006.
- [122] Ines Fey and Ingo Stürmer. Quality Assurance Methods for Model-based Development: A Survey and Assessment. *SAE International World Congress, SAE Doc. 2007-01-0506*, 2007.
- [123] Florian Fieber, Nikolaus Regnat, and Bernhard Rumpe. Assessing usability of model driven development in industrial projects. *Proceedings of the 4th European Workshop on "From code centric to model centric software engineering: Practices, Implications and ROI" (C2M), Series WP09-07*, June 2009.

- [124] Nils Fischer. *Design of a Plug-and-Play Development Environment for Optimizing Avionics Systems Architectures*. Faculty Computer Science & Automation, University of Technology Ilmenau, July 2007. Diplomarbeit (degree dissertation, comparable to master thesis).
- [125] Nils Fischer and Horst Salzwedel. "Complex Networked Avionics Systems Design at Early Conceptual Architecture Level". *Dresdner Arbeitstagung Schaltungs- und Systementwurf (DASS)*, 18.-19. May 2011.
- [126] Nils Fischer and Horst Salzwedel. "Overcoming The Generation Gap in Aircraft Designs with Executable Specifications". *30th Digital Avionics Systems Conference (DASC 2011), IEEE/AIAA*, 19.-22. July 2011.
- [127] Nils Fischer and Horst Salzwedel. "Validating Avionics Conceptual Architectures with Executable Specifications". *International Symposium on Models and Modeling Methodologies in Science and Engineering: MMMse 2011*, 19.-22. July 2011.
- [128] Nils Fischer and Horst Salzwedel. "Validating Avionics Conceptual Architectures with Executable Specifications". *Journal on Systemics, Cybernetics and Informatics (JSCI), Volume 10 / Number 4 / Year 2012*, pages 46–55, August 2012.
- [129] Nils Fischer and Mark Wiegmann. "Komplexe Flugzeugsysteme modellbasiert entwerfen - Entwurfsunsicherheiten durch Modellbildung minimieren". In *Proceedings of the Embedded Software Engineering Kongress (ESE 2013)*, Sindelfingen, Germany, 1-5 December 2013.
- [130] Nils Fischer and Mark Wiegmann. "More Efficient Design of Complex Aircraft Systems - Early Reduction of Design Uncertainty with Model-based Engineering". *2nd Consortium on Applied Research and Professional Education (CARPE) Conference*, 4-6 November 2013.
- [131] Wolfgang Fleisch. "Validierung komponentenbasierter Software für eingebettete Echtzeitsysteme". *VDI/VDE GMA-Kongreß'98, Mess- und Automatisierungstechnik - Neue Entwicklungen, Technologien, Anwendungen*, June 1998.
- [132] Wolfgang Fleisch. "Applying Use Cases for the Requirements Validation of Component-Based Real-Time Software". *Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'99)*, May 1999.
- [133] Wolfgang Fleisch. *Validierung komponentenbasierter Software für Echtzeitsysteme*. Shaker Verlag GmbH, 1st edition, 2003. IAS-Forschungsberichte, Band 1/2003, Forschungsbericht Institut für Automatisierungs- und Softwaretechnik Universität Stuttgart, Publisher: Prof. Dr.-Ing. Dr. h.c. P. Göhner.
- [134] Filippo De Florio. *Airworthiness: An Introduction to Aircraft Certification*. Elsevier Ltd, Oxford, 2010.

- [135] European Organization for Civil Aviation Equipment (EUROCAE). ED-12B - Software considerations in airborne systems and equipment certification, Including Amendment No1..19 October 1999. <http://www.eurocae.net>, 1992. obtainable online.
- [136] European Organization for Civil Aviation Equipment (EUROCAE). ED-80 - Design Assurance Guidance for Airborne Electronic Hardware. <http://www.eurocae.net>, 2000. obtainable online.
- [137] European Organization for Civil Aviation Equipment (EUROCAE). ED-202 - Airworthiness Security Process Specification. <http://www.eurocae.net>, 2010. obtainable online.
- [138] European Organization for Civil Aviation Equipment (EUROCAE). ED-79A - Guidelines for Development of Civil Aircraft and Systems. <http://www.eurocae.net>, 2010. obtainable online.
- [139] European Organization for Civil Aviation Equipment (EUROCAE). ED-14G - Environmental Conditions and Test Procedures for Airborne Equipment. <http://www.eurocae.net>, 2011. obtainable online.
- [140] European Organization for Civil Aviation Equipment (EUROCAE). ED-216 - Formal Methods supplement to ED-12C and ED-109A. <http://www.eurocae.net>, 2012. obtainable online.
- [141] Kevin Forsberg and Harold Mooz. The Relationship of System Engineering to the Project Cycle. *National Council For Systems Engineering (NCOSE) First Annual Conference - 'A Commitment to Success'*, 20-23 October 1991.
- [142] Tony Freeth. Die Entschlüsselung eines antiken Computers. *Spektrum der Wissenschaft, Spektrum der Wissenschaft Verlagsgesellschaft mbH, Verlagsgruppe Georg von Holtzbrinck*, pages 62–70, May 2010.
- [143] Ulrich Freund. *Integrierte Simulationstechniken auf Systemebene*. Logos Verlag Berlin, 1999.
- [144] Sanford Friedenthal, Regina Greigo, and Mark Sampson. INCOSE Model Based Systems Engineering (MBSE) Initiative - MBSE Roadmap. *The Seventeenth International Symposium of the International Council on Systems Engineering (INCOSE) Symposium 2007*, June 24-29 2007.
- [145] Michael C. Fu. A Tutorial Review of Techniques for Simulation Optimization. In *Proceedings of the 1994 Winter Simulation Conference*, pages 149–156, Dec 1994.
- [146] Daniel Galin. *Software Quality Assurance: From Theory to Implementation*. Pearson/Addison-Wesley, 2003.
- [147] Arthur Gill. *Introduction to the Theory of Finite-state Machines*. McGraw-Hill Inc.,US, 1962.
- [148] Arnodl Lewis Glass and Keith James Holyoak. *Cognition. 2nd Edition*. Random House USA Inc, 6 1988.

- [149] Wolfgang Gleine. Systemvalidierung and -verifizierung mithilfe von Softwareagenten. *Forschung Neues Fliegen - Competence Centre Neues Fliegen*, 1:12–13, 2012. PPN: 73859685X, Gemeinsamer Bibliotheksverbund GBV, Germany).
- [150] Phillipp Grieb. *Digital Prototyping*. Hanser Fachbuchverlag, 2010.
- [151] The Standish Group. *Chaos Report*. The Standish Group, 1994. 1st edition, updated annually.
- [152] The Standish Group. Big Bang Boom. https://www.standishgroup.com/sample_research_files/BigBangBoom.pdf, 2014. Copyright © 2014, The Standish Group International, Inc., available online, retrieved 01/06/2016.
- [153] Michael Gubisch. Raising the Bar, Made to Measure and Comfort Zone. *Flight International*, 181:33–37, 20-26 March 2012.
- [154] Reinhard Haberfellner, Peter Nagel, Mario Becker, Alfred Büchel, and Heinrich von Massow. *Systems Engineering - Methodik und Praxis*. Editor: W.F. Daenzer, F. Huber, Verlag Industrielle Organisation Zürich, Orell Füssli Verlag, 2002.
- [155] Anthony Hall. Seven Myths of Formal Methods. *IEEE Software*, 7(5):11–19, September 1990.
- [156] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
- [157] John Hines. Design - Why We Don't Do It Right. In *9th International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2001), keynote speech, 15-18 August 2001, Cincinnati, OH, (USA)*, page 169, 2001.
- [158] Martin Hinsch. *Industrielles Luftfahrtmanagement: Technik und Organisation luftfahrttechnischer Betriebe*. Springer Verlag Berlin Heidelberg, 2010.
- [159] Hartmut Hintze and Ralf God. Ansatz für einen Systems Security Engineering Prozess zur Entwicklung eines Kabinenmanagementsystems der nächsten Generation. *Deutscher Luft- und Raumfahrtkongress (DLRK 2012)*, 10-12 September 2012.
- [160] Hartmut Hintze, Andreas Tolksdorf, and Ralf God. Cabin Core System – a Next Generation Platform for Combined Electrical Power and Data Services. *3rd International Workshop on Aircraft System Technologies (AST 2011)*, March 31 - April 1 2011.
- [161] Jon Holt, Simon A. Perry, and Mike Brownsword. *Model-Based Requirements Engineering (Iet Professional Applications of Computing)*. The Institution of Engineering and Technology, pages 50 et seq., 95 et seq., 113 et seq., 148 et seq., 2012.

- [162] Patrick T. Homer and Richard D. Schlichting. Supporting Heterogeneity and Distribution in the Numerical Propulsion System Simulation Project. *Proceedings of the 2nd International Symposium on High Performance Distributed Computing*, pages 187–195, 20–23 July 1993.
- [163] Hendrik Härter and Franz Graser. Der Software-User als (un)bekanntes Wesen. *Elektronikpraxis, Embedded Software Engineering Magazin*, pages 18–19, December 2013.
- [164] International Business Machines Corporation (IBM). IBM Rational DOORS DLX Reference Manual Release 9.3. Software Manual, 2011.
- [165] Joint Task Force Transformation Initiative. *Guide for Conducting Risk Assessments, Information Security, NIST Special Publication 800-30 Revision 1*. National Institute of Standards & Technology (NIST), U.S Department of Commerce, Computer Security Division, Information Technology Laboratory, Gaithersburg, MD 20899-8930, September 2012.
- [166] American National Standards Institute and Electronic Industries Alliance. ANSI-EIA-649-A Standard: National Consensus Standard for Configuration Management. Standard, obtainable from TechAmerica (copyright), 2001.
- [167] SAE International. Architecture Analysis and Design Language (AADL). Standard definition, SAE Document AS-5506, Version: 2004-11-05, 2004.
- [168] Douglas A. Irwin and Nina Pavcnik. Airbus versus Boeing revisited: international competition in the aircraft market. *Journal of International Economics*, 64:223–245, 2004.
- [169] Bill Jackson, John Griggs, Ken Costello, Dan Solomon, Marcus Fisher, Stephanie Ferguson, and Gregory Blaney. Independent Verification and Validation, IVV 09-1, Revision: L. National Aeronautics and Space Administration (NASA), Independent Verification & Validation Program, February 2nd, 2009.
- [170] Michael A. Jackson. *System Development*. Prentice-Hall, 1983.
- [171] Michael A. Jackson. *Software Requirements and Specifications: A Lexicon of Software Practice, Principles and Prejudices*. Addison Wesley, ACM Press Book, 1995.
- [172] Michael A. Jackson. *Problem Frames and Methods. Analysing and Structuring Software Development Problems*. Addison Wesley, ACM Press Book, 2000.
- [173] Ivar Jacobson, Magnus Christerson, and Patrik Jonsson. *Object-Oriented Software Engineering: A Use CASE Approach*. Addison-Wesley Longman, Amsterdam, 1 edition, 7 1992.
- [174] Mario Jeckle, Chris Rupp, Jürgen Hahn, Barbara Zengler, and Stefan Queins. *UML 2 glasklar*. Carl Hanser Verlag GmbH & CO, 2003.
- [175] Asawaree Prabhakar Kalavade. *System-Level Codesign of Mixed Hardware-Software Systems*. PhD thesis, University of California, Berkeley, 1995. Ph. D. Dissertation.

- [176] Noriaki Kano, Nobuhiko Seraku, Fumio Takahashi, and Shin ichi Tsuji. Attractive Quality and Must-Be Quality. *Journal of the Japanese Society for Quality Control*, 14(2):39–48, April 1984.
- [177] Supaporn Kansomkeat, Phachayanee Thiket, and Jeff Offutt. Generating test cases from UML activity diagrams using the Condition-Classification Tree Method. In *2nd International Conference on Software Technology and Engineering (ICSTE)*, volume 1, pages V1–62–V1–66, Oct 2010.
- [178] Immanuel Kant. *Kritik der reinen Vernunft*. Johann Friedrich Hartknoch, 1781.
- [179] Anthony Kenny. *Rationalism, Empiricism and Idealism: British Academy Lectures on the History of Philosophy*. Oxford University Press, 1986.
- [180] Matthias Kühn, Tommy Baumann, and Horst Salzwedel. Genetic Algorithm for Process Optimization in Hospitals. In *26th European Conference on Modelling and Simulation, ECMS 2012, Koblenz, Germany, May 29 - June 1, 2012*, pages 103–107, 2012.
- [181] Sascha Kirstan. *Kosten und Nutzen modellbasierter Entwicklung eingebetteter Softwaresysteme im Automobil*. Dissertation, Verlag Dr. Hut, 2011.
- [182] Sascha Kirstan and Jens Zimmermann. Evaluating Costs and Benefits of Model-based Development of Embedded Software Systems in the Car Industry - Results of a Qualitative Case Study. *The Fifth Workshop "From code centric to model centric: Evaluating the effectiveness of MDD (C2M:EEMDD)"*, 15 - 18 June 2010.
- [183] Stephan Koenen. Investigation of the Economic Impact of Changes in Regulatory Bodies Using the Examples of the A320 and A350 Aircraft Families. *Master Thesis, WHU - Otto Beisheim School of Management*, 2012.
- [184] Stephen M. Kosslyn. Imagery in Learning. In Michael S. Gazzaniga, editor, *Perspectives in Memory Research*, pages 245–275. MIT Press, 1988.
- [185] Gerald Kotonya and Ian Sommerville. *Requirements Engineering: Processes and Techniques*. John Wiley & Sons, Reprinted 2000, 2001, pages 4 ff., 1998.
- [186] Debasish Kundu and Debasis Samanta. A Novel Approach to Generate Test Cases from UML Activity Diagrams. *Journal of Object Technology*, 8(3):65–83, May-June 2009.
- [187] Gérard Le Lann. The Ariane 5 Flight 501 Failure - A Case Study in System Engineering for Computing Systems. *Institut National de Recherche en Informatique et en Automatique (INRIA) Research Report RR-3079*, 1996.
- [188] Peter E. Lauer. *Functional Programming, Concurrency, Simulation and Automated Reasoning - International Lecture Series 1991 / 1992 McMaster University, Hamilton, Ontario, Canada*. Lecture Notes in Computer Science Volume 693, Springer Verlag, 1993.

- [189] Odile Laurent. Using formal methods and testability concepts in the Avionics Systems Validation and Verification (V&V) process. *Third International Conference on Software Testing, Verification and Validation (ICST 2010)*, pages 1–10, 6-10 April 2010.
- [190] Jonah Z. Lavi and Joseph Kudish. *Systems Modeling and Requirements Specification Using ECSAM: An Analysis Method for Embedded and Computer-Based Systems*. Dorset House, 1st edition, 2004.
- [191] Jonah Z. Lavi and Joseph Kudish. Systems modeling & requirements specification using ECSAM: an analysis method for embedded & computer-based systems. *Innovations in Systems and Software Engineering*, 1(2):100–115, 2005.
- [192] Averill M. Law and Michael G. McComas. Simulation Software for Communications Networks: The State of the Art. *IEEE Communications Magazine*, 32(3), March 1994.
- [193] Clive Lee. *Introduction to DO-25*. ERA Technology Report, , as part of the Avionics Systems Standardization Committee (ASSC), DO178b-DO-254 Seminar, ERA Technology Ltd, 2006.
- [194] Edward A. Lee. Overview of the Ptolemy Project. Technical Memorandum No. UCB/ERL M03/25, 2004. University of California, Berkeley, CA, 94720, USA, July 2nd, 2003.
- [195] Eckard Lehmann. *Time Partition Testing - Systematischer Test des kontinuierlichen Verhaltens von eingebetteten Systemen*. Dissertation, Faculty Electrical Engineering and Computer Science, Berlin University of Technology, 2004.
- [196] Oliver Lemke. *Modellbasierte Anforderungsspezifikation sicherheitskritischer Systeme im Bahnbereich – Beschreibungsmittel und ihre Anwendung*. PhD thesis, Fakultät Architektur, Bauingenieurwesen und Umweltwissenschaften der Technischen Universität Carolo-Wilhelmina zu Braunschweig, 2010. Ph. D. Dissertation.
- [197] B. Lewis and P. H. Feiler. Incremental Verification and Validation of System Architecture for Software Reliant Systems Using AADL(Architecture Analysis & Design Language). *Layered Assurance Workshop, Software Engineering Institute Carnegie Mellon University, Pittsburgh, USA*, December 6, 2010.
- [198] Thomas Volker Liebezeit. *Missionsbezogener modellgestützter Entwurf mobiler automatischer Systeme am Beispiel eines autonomen Unterwasserfahrzeugs*. Dissertation, Faculty Computer Science & Automation, Ilmenau University of Technology, 2005.
- [199] Otto Lilienthal. *Der Vogelflug als Grundlage der Fliegekunst*. R. Gaertners Verlagsbuchhandlung Berlin, 1889.
- [200] Otto Lilienthal. *Birdflight as the Basis of Aviation*. Longmans, Green, and Co.; 39 Paternoster Row, London, New York, Bombay and Calcutta, 1911.

- [201] Otto Lilienthal. Letter to Moritz von Egidy. <http://www.lilienthal-museum.de/olma/l1852.htm>, ca. January 1894. Otto-Lilienthal-Museum, Anklam, Germany, available online, retrieved 09/24/2016.
- [202] Wang Linzhang, Yuan Jiesong, Yu Xiaofeng, Hu Jun, Li Xuandong, and Zheng Guoliang. Generating test cases from UML activity diagram based on Gray-box method. In *Software Engineering Conference, 2004. 11th Asia-Pacific*, pages 284–291, Nov 2004.
- [203] Guideline Verein Deutscher Ingenieure (VDI) VDI-Fachbereich Technische Logistik. VDI-Richtlinie 2519 Blatt 1 - Procedures for the compilation of tender and performance specifications. DIN 69901-5:2009-01, 2009. obtainable from Verein Deutscher Ingenieure (VDI).
- [204] Achim Loock, Markus Töppel, and Andreas Prücklmeier. Integration von Basis- und Kabinensystemen (INKABA): Abschlussbericht des Verbundvorhabens ; Luftfahrtforschungsprogramm LUFO IV. final project report (public), Reportnr. / Förderkennzeichen: 20K0702A, 2008. Technische Informationsbibliothek und Universitätsbibliothek Hannover (TIB/UB), DOI: 10.2314/GBV:604071663.
- [205] Visual Paradigm International Ltd. Visual Paradigm User’s Guide. <http://www.visual-paradigm.com/support/documents/vpuserguide.jsp>, 2014. User Manual, Last Modified: Dec 23, 2014, retrieved 01/05/2015.
- [206] Lufthansa, Bruno Piquet (pub.), and Kenneth Johnson (ed.). In flight entertainment - How in flight entertainment has changed! *FAST 39 - Flight Airworthiness Support Technology*, 39:40, December 2006. available online, retrieved 07/11/2014.
- [207] Jan Lunze. *Ereignisdiskrete Systeme: Modellierung und Analyse dynamischer Systeme mit Automaten, Markovketten und Petrinetzen*. Oldenbourg Wissenschaftsverlag, 2 edition, 2012.
- [208] Luqi and Joseph A. Goguen. Formal methods: promises and problems. *IEEE Software*, 14(1):73–85, January / February 1997.
- [209] Imran Mahmud and Vito Veneziano. Mind-mapping: An Effective Technique to Facilitate Requirements Engineering in Agile Software Development. In *14th International Conference on Computer and Information Technology (ICCIT 2011)*, pages 157–162, Dec 2011.
- [210] N.A.M. Maiden. CREWS-SAVRE: Scenarios for Acquiring and Validating Requirements. *Automated Software Engineering*, 5(4):419–446, 1998.
- [211] M. W. Maier. Architecting Principles for Systems-of-Systems Engineering. *Systems Engineering*, 1(4):267–284, 1998.
- [212] Tets Maniwa. Focus Report: Electronic System-Level (ESL) Tools. Chip Design Magazine, 2004. issue April/May 2004.
- [213] Jo Marchant. *Decoding the Heavens: Solving the Mystery of the World’s First Computer*. Random House UK, 2009.

- [214] Robert C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall International, 2002.
- [215] Stephan Marwedel, Nils Fischer, and Horst Salzwedel. Improving the design quality of complex networked systems using a modelbased approach. *Third International Conference on Modelbased Systems Engineering (IC-MBSE 2010), IEEE/INCOSE*, 27.-28. September 2010.
- [216] Stephan Marwedel, Nils Fischer, and Horst Salzwedel. Improving Performance and Reliability Assessments of Avionics Systems. *30th Digital Avionics Systems Conference (DASC 2011), IEEE/AIAA*, 19.-22. July 2011.
- [217] Timothy McPhillips, Shawn Bowers, Daniel Zinn, and Bertram Ludäscher. Scientific workflow design for mere mortals. *Future Generation Computer Systems*, 25(5):541 – 551, 2009.
- [218] Stephen Mellor and Leon Starr. *Six Lessons Learned Using MDA*. Springer-Verlag Berlin, Lecture Notes in Computer Science, Volume 3297, UML Modeling Languages and Applications, pages 198-202, 2005.
- [219] Stephen J. Mellor and Marc J. Balcer. *Executable UML: A Foundation for Model-Driven Architecture*. Addison Wesley Pub Co Inc, 2002.
- [220] Arno Meyna and Bernhard Pauli. *Zuverlässigkeitstechnik: Quantitative Bewertungsverfahren*. Carl Hanser Verlag GmbH & CO. KG, 2010.
- [221] Lars Mieritz and Roger Fulton. The Business Value of IT: New Answers to an Old Problem. In *Gartner Symposium/ITxpo, European Symposium, 7-11 November 2005, Cannes, France*, 2005.
- [222] Inc. MLDesign Technologies. MLDesigner Documentation. <http://www.mldesigner.com/fileadmin/assets/MLDesigner/Manual/manual.pdf>, 2010. User manual, Version 2.8 - February 26, available online, retrieved 01/16/2013.
- [223] Parastoo Mohagheghi and Vegard Dehlen. *Where Is the Proof? - A Review of Experiences from Applying MDE in Industry*. Springer-Verlag Berlin, ECMDA-FA '08 Proceedings of the 4th European conference on Model Driven Architecture: Foundations and Applications, appears in: Lecture Notes in Computer Science, 2008.
- [224] Ian Moir, Allan Seabridge, and Malcolm Jukes. *Civil Avionics Systems*. John Wiley and Sons Ltd., pages 160 et seq., 2 edition, 10 2013.
- [225] Ian Moir and Allan G. Seabridge. *Design and Development of Aircraft Systems: An Introduction*. American Institute of Aeronautics and Astronautics (AIAA), Education Series, pages 9 et seq., 21 et seq., 39 et seq., 65 et seq., 71, 77 et seq., 90 et seq., 111 et seq., 2004.
- [226] Gordon E. Moore. Cramming more components onto integrated circuits. In: *Electronics*, 38(8):114–117, 1965.

- [227] NDIA. Modeling & Simulation Committee. http://www.ndia.org/Divisions/Divisions/SystemsEngineering/Pages/Modeling_and_Simulation_Committee.aspx, 2013. National Defense Industrial Association (NDIA) Systems Engineering Division.
- [228] Uwe Neumann, Ben Holert, and Udo Carl. Signal- und modellbasierte Konzepte zur elektronischen Lastbegrenzung im Antriebsstrang von Hochauftriebssystemen. *Proceedings of the Deutscher Luft- und Raumfahrtkongress 2003 (DLRK 2003)*, 17-20 November 2003.
- [229] Uwe Neumann, Ben Holert, and Udo Carl. Für eine sichere Landung. Simulation von Landeklappenantriebssystemen, using the example of the Airbus A340. <http://doku.b.tu-harburg.de/volltexte/2006/271/>, 2004. Article from the magazine "Antriebstechnik" no. 43 (2004), pp. 70-74, available online, retrieved 11/12/2011.
- [230] David M. Nicol, Daniel L. Palumbo, and Michael L. Ulrey. A Graphical Model-Based Reliability Estimation Tool and Failure Mode & Effects Simulator. *Proceedings of the Annual Reliability and Maintainability Symposium*, pages 74-81, 16-19 January 1995.
- [231] Norm. Quality management systems - Fundamentals and vocabulary. DIN EN ISO 9000:2005-12, 2005. obtainable from Beuth Verlag GmbH.
- [232] Norm. Quality management systems - Requirements. DIN EN ISO 9001:2008, 2008. obtainable from Beuth Verlag GmbH.
- [233] Norm. Project management - Project management systems - Part 5: Concepts. DIN 69901-5:2009-01, 2009. obtainable from Beuth Verlag GmbH.
- [234] Norm. Quality Management Systems - Requirements for Aviation, Space and Defense Organizations. DIN EN 9100:2010-07, 2010. obtainable from Beuth Verlag GmbH.
- [235] Guy Norris and Mark Wagner. *Boeing 777 (Enthusiast Color)*. Motorbooks International, 1996.
- [236] Guy Norris and Mark Wagner. *Modern Boeing Jetliners*. Motorbooks International, 1999.
- [237] Guy Norris and Mark Wagner. *Airbus A380: Superjumbo of the 21st Century*. Zenith Press, pages 7, 16-18, 2005.
- [238] Society of Automotive Engineers (SAE International). ARP4754a - Guidelines for Development of Civil Aircraft and Systems, Revision Number: A. <http://standards.sae.org/arp4754a/>, 2010. obtainable online.
- [239] Department of Defense (DoD). MIL-STD-882-B - Military Standard System Safety Program Requirements, AMSC Number F3329. Military Standard, Washington, DC, United States of America, 1984.
- [240] Department of Defense (DoD). MIL-HDBK-61 - Military Handbook: Configuration Management Guidance. Department of Defense Handbook, Washington, DC, United States of America, 2001.

- [241] Department of Defense (DoD). DOD Architecture Framework (DoDAF), V1.5, Volumes I, II, and III, Version 1.5 ed. United States of America, 2007.
- [242] Under Secretary of Defense for Acquisition and Technology. Modeling and Simulation (M&S) Master Plan. <http://www.dtic.mil/whs/directives/corres/pdf/500059p1.pdf>, 1995. U.S. Department of Defense (DoD), DoD Phamplet 5000.59-P, retrieved 01/11/2013.
- [243] The Institute of Electrical and Electronic Engineers. *IEEE Std 610 - IEEE Standard Computer Dictionary, A Compilation of IEEE Standard Computer Glossaries*. (IEEE), 1990.
- [244] The Institute of Electrical and Electronic Engineers (IEEE). *IEEE Std 829-2008 - IEEE Standard for Software and System Test Documentation*. (IEEE), 2008.
- [245] Software Engineering Standards Committee of the IEEE Computer Society. IEEE Standard for Software Safety Plans. IEEE Std 1228-1994, 1994. The Institute of Electrical and Electronics Engineers, Inc.
- [246] Systems Engineering Vision Working Group of the International Council on Systems Engineering (INCOSE). Systems Engineering Vision 2020. *International Council on Systems Engineering (INCOSE), Document No.: INCOSE-TP-2004-004-02, Version 2.03*, September 2007.
- [247] Object Management Group (OMG). UMLTMProfile for Schedulability, Performance, and Time Specification. *OMG Document Number: formal/05-01-02, Version 1.1, Standard Document URL: <http://www.omg.org/spec/SPTP/>, pages 2-6 ff., 3-6 ff., 3-26 ff., 5-2 ff.*, January 2005.
- [248] Object Management Group (OMG). Action Language for Foundational UML (Alf). *OMG RFP, OMG Document Number: formal/2013-09-01, Version 1.0.1, Standard Document URL: <http://www.omg.org/spec/ALF/1.0.1/PDF/>*, October 2013.
- [249] Object Management Group (OMG). Semantics of a Foundational Subset for Executable UML Models (fUML). *OMG RFP, OMG Document Number: formal/2016-01-05, Version 1.2.1, Standard Document URL: <http://www.omg.org/spec/FUML/1.2.1/PDF/>*, January 2016.
- [250] SE Handbook Working Group International Council on Systems Engineering (INCOSE). Systems Engineering Handbook, a Guide for System Life Cycle Processes and Activities. *International Council on Systems Engineering (INCOSE), Document No.: INCOSE-TP-2003-002-03.2, Edited by Cecilia Haskins, CSEP*, January 2010.
- [251] Technical Board International Council on Systems Engineering (INCOSE). Systems Engineering Handbook, a 'What to' Guide for all SE Practitioners. *International Council on Systems Engineering (INCOSE), Document No.: INCOSE-TP-2003-016-02, Version 2a*, June 2004.
- [252] John K. Ousterhout, Ken Jones, and Eric Foster-Johnson. *Tcl and the Tk Toolkit*. Addison Wesley Pub Co Inc, 2 edition, 9 2009.

- [253] Jerry Overton, Jon G Hall, and Lucia Rapanotti. Middle-out design: A proposed best-practice for GEOSS design. Technical Report 2010/10, Department of Computing Faculty of Mathematics, Computing and Technology, The Open University, May 2010. ISSN 1744-1986.
- [254] Alexander Pacholik. *Entwicklung und Gegenüberstellung von Methoden zur automatisierten Verifikation von ausführbaren Systemspezifikationen*. Dissertation, University of Technology Ilmenau, 2010.
- [255] Nils Paluch. *Anwendung von Co-Design für verteilte Echtzeitsysteme Einführung und Ausblicke am Beispiel des Tiefflugsystems TMLLF an Bord des Airbus A400M*. Master Thesis (Diplomarbeit), Faculty Computer Science & Automation, Ilmenau University of Technology, 2004.
- [256] Pascal Pampagnin, Pierre Moreau, Remy Maurice, and David Guihal. Model driven hardware design: One step forward to cope with the aerospace industry needs. *Forum on Specification, Verification and Design Languages (FDL 2008)*, pages 179 – 184, 23-25 September 2008.
- [257] Vikas V. Patel, John D. McGregor, and Sebastien Goasguen. SysML-based Domain-specific Executable Workflows. In *4th Annual IEEE Systems Conference*, pages 505–509, April 2010.
- [258] Trevor W. Pearce. Simulation-Driven Architecture in the Engineering of Real-Time Embedded Systems. In *24th IEEE Real-Time Systems Symposium, work-in-progress session (RTSS 2003)*, pages 332–339, Cancun Mexico, 12 2003.
- [259] Dominic Perry. EC225 grounding cost Bond £4.1 million in three months. <http://www.flightglobal.com/news/articles/ec225-grounding-cost-bond-1634.1-million-in-three-392079/>, 2013. Flightglobal, 24 Oct 2013, retrieved 10/10/2014.
- [260] Dominic Perry. False alarms drove EC225 ditchings. *Flight International*, 183:9, 26 March - 1 April 2013. related online report: 'Incorrect specification caused EC225 emergency lube fault' by Dominic Perry, 21 March 2013, <http://www.flightglobal.com/news/articles/incorrect-specification-caused-ec225-emergency-lube-383682/>.
- [261] Gregory D. Peterson and John W. Hines. Advanced avionics system development: achieving systems superiority through design automation. *IEEE Aerospace Conference 1998*, pages 231 – 238, 21-28 March 1998.
- [262] Alexander K. Petrenko and Olga L. Petrenko. Formal Methods and Innovation Economy: Facing New Challenges. *Sixth IEEE International Conference on Software Engineering and Formal Methods (SEFM '08)*, pages 367 – 371, 10-14 November 2008.
- [263] John Pfoestl. Requirements Based Engineering for Aerospace Certification Projects - RBE-Handout Day One. http://www.n-w-c.de/files/RBE-Handout-DAY1_.pdf, 2010. Preparation of seminars and presentations on ABD0100, GRESS, RTCA/DO-178, DO-254 and requirements based engineering, Handout Day 1, available online, retrieved 11/30/2012.

- [264] John Pfoestl. Requirements Based Engineering for Aerospace Certification Projects - RBE-Handout Day Two. http://www.n-w-c.de/files/RBE-Handout-DAY2_ABR.pdf, 2010. Preparation of seminars and presentations on ABD0100, GRESS, RTCA/DO-178, DO-254 and requirements based engineering, Handout Day 2, available online, retrieved 11/30/2012.
- [265] Dan Pilone and Neil Pitman. *UML 2.0 in a Nutshell (A Desktop Quick Reference)*. O'Reilly & Associates, pages 77 et seq., 1 edition, 6 2005.
- [266] Tom Pleasant. Going Up? *Aircraft Cabin Management*, 2:8–13, April 2013.
- [267] Sabri Pllana, Thomas Fahringer, Johannes Testori, Siegfried Benkner, and Ivona Brandic. Towards an UML Based Graphical Representation of Grid Workflow Applications. In Marios D. Dikaiakos, editor, *Grid Computing*, volume 3165 of *Lecture Notes in Computer Science*, pages 149–158. Springer Berlin Heidelberg, 2004.
- [268] Klaus Pohl and Chris Rupp. *Requirements Engineering Fundamentals: A Study Guide for the Certified Professional for Requirements Engineering Exam (Foundation Level, IREB compliant)*. Rocky Nook, pages 19-39, 67-99, 1 edition, 5 2011.
- [269] Nadège Pontisso and David Chemouil. TOPCASED Combining Formal Methods with Model-Driven Engineering. *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006)*, pages 359 – 360, 18-22 September 2006.
- [270] Colin Potts. Using Schematic Scenarios to Understand User Needs. In *Proceedings of the 1st Conference on Designing Interactive Systems: Processes, Practices, Methods, & Techniques*, DIS '95, pages 247–256, New York, NY, USA, 1995. ACM.
- [271] Defense Acquisition University Press. System Engineering Fundamentals. <http://www.dau.mil/pubscats/PubsCats/SEFGuide%2001-01.pdf>, 2001. Guideline, U.S. Department of Defense (DoD), retrieved 01/11/2013.
- [272] Jarmo Prokkola. Simulations and Tools for Telecommunications 521365S: OPNET - Network Simulator, VTT Technical Research Centre of Finland. *Lecture at Tietotalo, University of Oulu*, 19 April 2006.
- [273] Inc.) Radio Technical Commission for Aeronautics (RTCA. DO-178B - Software Considerations in Airborne Systems and Equipment Certification. <http://www.rtca.org>, 1992. obtainable online.
- [274] Inc.) Radio Technical Commission for Aeronautics (RTCA. DO-254 - Design Assurance Guidance for Airborne Electronic Hardware. <http://www.rtca.org>, 2000. obtainable online.
- [275] Inc.) Radio Technical Commission for Aeronautics (RTCA. DO-160G - Environmental Conditions and Test Procedures for Airborne Equipment. <http://www.rtca.org>, 2010. obtainable online.

- [276] Inc.) Radio Technical Commission for Aeronautics (RTCA. DO-326 - Airworthiness Security Process Specification. <http://www.rtca.org>, 2010. obtainable online.
- [277] Inc.) Radio Technical Commission for Aeronautics (RTCA. DO-178C - Software Considerations in Airborne Systems and Equipment Certification. <http://www.rtca.org>, 2011. obtainable online.
- [278] Inc.) Radio Technical Commission for Aeronautics (RTCA. DO-278A - Software Integrity Assurance Considerations for Communication, Navigation, Surveillance and Air Traffic Management (CNS/ATM) Systems. <http://www.rtca.org>, 2011. obtainable online.
- [279] Inc.) Radio Technical Commission for Aeronautics (RTCA. DO-331 - Model-Based Development and Verification Supplement to DO-178C and DO-278A. <http://www.rtca.org>, 2011. obtainable online.
- [280] Inc.) Radio Technical Commission for Aeronautics (RTCA. DO-333 - Formal Methods Supplement to DO-178C and DO-278A. <http://www.rtca.org>, 2011. obtainable online.
- [281] Holger Rath, Gunnar Schorcht, and Horst Salzwedel. Simulationsumgebung für Bordnetze - Bordnetz-Spezifikationen modellieren und validieren. *Hanser Automotive, Carl Hanser Verlag München*, 2006.
- [282] Michael Rath. *Generierung von Architekturmodellen aus Verhaltensmodellen basierend auf dem Network Block Set im MLDesigner*. Student research project, Faculty Computer Science & Automation, Ilmenau University of Technology, 2007.
- [283] Samik Raychaudhuri. Introduction to Monte Carlo Simulation. In *Winter Simulation Conference 2008 (WSC 2008)*, pages 91–100, Dec 2008.
- [284] Kåre Sjölander. The Snack Sound Toolkit. <http://www.speech.kth.se/snack/>, 2004. Snack Sound Toolkit developer homepage, KTH Royal Institute of Technology Stockholm, retrieved 04/30/2015.
- [285] Casey B. Reardon, Ian A. Troxel, and Alan D. George. Virtual prototyping of WDM avionics networks. In *Avionics Fiber-Optics and Photonics, 2005. IEEE Conference*, pages 17–18, Sept 2005.
- [286] David Redman, Donald Ward, John Joseph Chilenski, and Greg Pollari. Virtual Integration for Improved System Design. *Proceedings of the Analytic Virtual Integration of Cyber-Physical Systems Workshop (AVICPS)*, November 30 2010.
- [287] Jörg Reitmann. Neue Technologien und Trends in der Kabinenkommunikation. Lecture at Praxis-Seminar Luftfahrt, Deutsche Gesellschaft für Luft- und Raumfahrt (DGLR), Verein Deutscher Ingenieure (VDI) Hamburger Bezirksverein, Arbeitskreis Luft- und Raumfahrt, 30. September 2004. Hamburg University of Applied Sciences (HAW-Hamburg).

- [288] Ronald Renner, Wolfram Kißling, and Barbara Zaunseder. *Rapid Prototyping: Unterstützung der kundenorientierten Produktentwicklung*. Vde-Verlag, 2000.
- [289] Joachim Reuter, Sven-Olaf Berkhahn, and Wolfgang Fischer. Technologie Trends in zukünftiger Passagierkabine. Public Lecture at Praxis-Seminar Luftfahrt, Deutsche Gesellschaft für Luft- und Raumfahrt (DGLR), Royal Aeronautical Society Hamburg Branch e.V., Verein Deutscher Ingenieure (VDI) Hamburger Bezirksverein, Arbeitskreis Luft- und Raumfahrt, 11 February 2010. Hamburg University of Applied Sciences.
- [290] Andreas Reuys, Erik Kamsties, Klaus Pohl, and Sacha Reis. Model-Based System Testing of Software Product Families. In *Advanced Information Systems Engineering*, volume 3520 of *Lecture Notes in Computer Science*, pages 519–534. Springer Berlin Heidelberg, 2005.
- [291] Suzana Kahn Ribeiro, Shigeki Kobayashi, Michel Beuthe, Jorge Gasca, David Greene, David S. Lee, Yasunori Muromachi, Peter J. Newton, Steven Plotkin, Daniel Sperling, Ron Wit, and Peter J. Zhou. Transport and its infrastructure. In *Climate Change 2007: Mitigation. Contribution of Working Group III to the Fourth Assessment Report of the Intergovernmental Panel on Climate Change*. <http://www.ipcc.ch/pdf/assessment-report/ar4/wg3/ar4-wg3-chapter5.pdf>, 2007. Cambridge University Press, Cambridge, United Kingdom and New York, NY, USA, available online, retrieved 07/14/2014.
- [292] Thomas Roßner, Christian Brandes, Helmut Götz, and Mario Winter. *Basiswissen Modellbasierter Test*. dpunkt Verlag, pages vii et seq., 35 et seq., 185 et seq., 1. edition, 8 2010.
- [293] Barry Rosenberg. Cabin Management Systems. *Avionics Magazine - Solutions for Global Airspace Electronics*, 34:26–30, May 2010.
- [294] R. K. Rosich. The Modeling Process - The Key Elements of the Modeling Plan. Technical report, Hughes Aircraft Company, Colorado Engineering Laboratories, 1992. 5 August, 16800 E. CentreTech Parkway, Aurora, CO, 80011, (303) 344-6371.
- [295] Lillian Røstad. An extended misuse case notation: Including vulnerabilities and the insider threat. *The Twelfth Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ'06) held in conjunction with CAiSE'06*, pages 33–43, 5-6 June 2006.
- [296] Andrea Rothman. Airbus Fell Short with 10 A380s in 2009. *Bloomberg Businessweek - Global Economics*, December 30 2009. available online, retrieved 07/23/2014.
- [297] Jean-Clause Roussel. AIRBUS Presentation - Benefits of Requirement Engineering with DOORS. http://www.incose.org/japan/ijc_data/20050822/IJC20050822_07.pdf, 22nd June 2005. Malmö, Sweden, TELELOGIC Capital Market Event, INCOSE publication, retrieved 11/22/2012.
- [298] Jerzy W. Rozenblit. Systems Design: A Simulation Modeling Framework. In Hessam S. Sarjoughian and François E. Cellier, editors, *Discrete Event Modeling and Simulation Technologies*, pages 107–129. Springer New York, 2001.

- [299] Jerzy W. Rozenblit. Complex Systems Design: A Simulation Modelling Approach. In *Proceedings of the International Workshop on Harbor, Maritime and Multimodal Logistics Modeling & Simulation HMS2003*, pages 17–20, September 2003.
- [300] Research Triangle Institute (RTI). *Planning Report 02-3, The Economic Impacts of Inadequate Infrastructure for Software Testing*. National Institute of Standards & Technology (NIST), U.S Department of Commerce, Technology Administration, Project Number 7007.011, May 2002.
- [301] Chris Rupp and die SOPHISTen. *Requirements-Engineering und -Management: Professionelle, iterative Anforderungsanalyse für die Praxis*. Carl Hanser Verlag GmbH & Co. KG, 5. aktualisierte und erweiterte auflage edition, 2009.
- [302] Johannes Ryser and Martin Glinz. A Practical Approach to Validating and Testing Software Systems Using Scenarios. In *3rd International Software Quality Week Europe (QWE '99)*, 1999.
- [303] Esterel Technologies SA. *Methodology Handbook: Efficient Development of Safe Avionics Display Software with DO-178B Objectives using Esterel SCADE*. Esterel Technologies, 2009. Revision: SDY-HB-DO178B - SDY/u3-KCG612.
- [304] Karl Sabbagh. Video series TCJE-1, PBS Video. In *21st Century Jet: The Building of the 777, Vol. 1 - To Design a Plane*, 1996.
- [305] Sadegh Sadeghipour. Testing Reactive Systems on the Basis of Formal Specifications. *Scientia Iranica - International Journal of Science and Technology, Computer Engineering*, 8(4):241–249, October 2001.
- [306] A. P. Sage and J. E. Armstrong. *Introduction to Systems Engineering*. John Wiley & Sons, 2000.
- [307] A. P. Sage and C. D. Cuppan. On the systems engineering and management of systems of systems and federations of systems. *Information, Knowledge, Systems Management*, 2(4):325–334, 2001.
- [308] H. Salzwedel, R. Calhoun, and Lt. P. Murdock. *Unbiased Transfer Alignment Filter Design for Air Launched Weapons*. National Aerospace and Electronics Conference, Dayton, Ohio, USA, 1985.
- [309] H. Salzwedel and J. Vincent. *Modeling, identification, and control of flexible aircraft*. Air Force Wright Aeronautical Laboratories report, no. AFWAL-TR-84-3032. Air Force Wright Aeronautical Laboratories, 1984.
- [310] Horst Salzwedel. Solving the Complexity Challenge in the Design of Electronic Systems. *Challenge-the-Expert, Silicon Valley World Internet Center, Palo Alto, California, USA*, March 20 2003.
- [311] Horst Salzwedel. Mission Level Design of Avionics. *AIAA-IEEE DASC 04 - The 23rd Digital Avionics Systems Conference 2004*, page 9D.2, October 2004.

- [312] Horst Salzwedel. Large Scale Networked System Simulation Using MLDesigner. *Sixth Biennial Ptolemy / Kepler Miniconference, Berkeley, California, USA*, May 12 2005.
- [313] Horst Salzwedel. Complex System Design Automation in the Presence of Bounded and Statistical Uncertainties. *52. Internationales Wissenschaftliches Kolloquium - IWK'2007*, September 2007.
- [314] Horst Salzwedel. Control Design of Aerospace and Automotive Systems - How to reduce risk in complex system design. *Symposium on Automotive/Avionics Systems Engineering SAASE 2009*, 14 October 2009.
- [315] Horst Salzwedel. Model Based Engineering of Aerospace Systems - How to Reduce Risk in Complex Systems Design. *Developing Aircraft PHotonic NETWORKS Project, DAPHNE MBE Tutorial Berlin*, December 10 2009.
- [316] Horst Salzwedel, Nils Fischer, and Tommy Baumann. Aircraft Level Optimization of Avionics Architectures. *27th Digital Avionics Systems Conference (DASC 2008), IEEE/AIAA*, 26.-30. October 2008.
- [317] Horst Salzwedel, Nils Fischer, and Gunar Schorcht. Moving Design Automation of Networked Systems to Early Vehicle Level Design Stages. *SAE 2009 World Congress*, 20.-23. April 2009.
- [318] Kristian Sandahl. Tutorial 3: Requirements Analysis and Model-Based Requirements Engineering. *4th MODPROD Workshop on Model-Based Product Development*, February 9-10 2010.
- [319] Airbus S.A.S. Airbus Global Market Forecast, Flying by Numbers 2015 - 2034. http://www.airbus.com/company/market/forecast/?eID=maglisting_push&tx_maglisting_pi1%5BdocID%5D=89373, 2015. Blagnac Cedex, France, available online, retrieved 22/10/2016.
- [320] Gerhard Schewe. Gabler Wirtschaftslexikon, Keyword: Workflow. <http://wirtschaftslexikon.gabler.de/Archiv/17045/workflow-v8.html>, 2014. Springer Gabler Verlag (publisher), Economy encyclopedia, Available online, retrieved 09/17/2014.
- [321] Ina Schieferdecker. Modellbasiertes Testen. *OBJEKTspektrum*, 1(3):39–45, 2007.
- [322] Bruno Schienmann. *Kontinuierliches Anforderungsmanagement . Prozesse-Techniken-Werkzeuge*. Addison-Wesley, 2001.
- [323] Gunar Schorcht. *Entwurf integrierter Mobilkommunikationssysteme auf Missionsebene*. Logos Verlag Berlin, 2000.
- [324] Gunar Schorcht, Ian Troxel, Keyvan Farhangian, Peter Unger, Daniel Zinn, Colin K. Mick, Alan George, and Horst Salzwedel. System-level simulation modeling with MLDesigner. In *11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems - MASCOTS 2003*, pages 207–212, Oct 2003.

- [325] Mario Schulz, Volker Zerbe, Kai Yang, and Armin Zimmermann. Optimization of Avionic System Architectures. *CEAS Aeronautical Journal*, 2(1-4):289–294, 2011.
- [326] Jochen Seemann and Jürgen Wolff von Gudenberg. *Software-Entwurf mit UML 2: Objektorientierte Modellierung mit Beispielen in Java*. Springer, 2. Aufl. 2006 edition, 4 2006.
- [327] K. Sam Shanmugan, Victor S. Frost, and William LaRue. A Block-Oriented Network Simulator (BONeS). *SAGE publications on behalf of the Society for Modeling & Simulation International (SCS), SIMULATION vol. 58 no. 2*, February 1992.
- [328] Manfred Sieber. Der Mensch steht im Mittelpunkt, User-Centred Design zur Entwicklung von Mensch-Maschine-Schnittstellen für die Kabinencrew. *VDI Mensch & Technik*, 17:21, 2011.
- [329] Axel Sikora and Rolf Drechsler. *Software-Engineering und Hardware-Design – Eine systematische Einführung*. Carl Hanser Verlag, 2002.
- [330] Guttorm Sindre and Andreas L. Opdahl. Capturing Security Requirements through Misuse Cases. *Norsk Informatikkonferanse*, 26-28 November 2001.
- [331] Guttorm Sindre and Andreas L. Opdahl. Eliciting security requirements with misuse cases. *Requirements Engineering*, 10(1):34–44, 2005.
- [332] Loren Slafer. Boeing Satellite Systems, Achieving Software Validation through Simulation. *Proceedings of the Applied Dynamics International User's Society (ADIUS) conference 2001 - ADIUS 2001: An Embedded Systems Odyssey*, 10 June 2001.
- [333] Gary Smith. EDA ESL Landscape 2012. http://www.garysmitheda.com/paper/ESL_Wall12.pdf, 2012. available online, retrieved 10/06/2014.
- [334] Gary Smith and Daya Nadamuni. ESL Landscape 2005. Gartner Dataquest, 2005.
- [335] Richard Smyth and Gerd Roloff. Best Practices for Systems Development and Integration. <http://www.finse.org/2006FallSeminar/Best%20pract%20Hel%20Summary%2026%20Oct%2006.pdf>, 2006. Helsinki, 26 October 2006, Fall Seminar of the Finnish Systems Engineering Association (FINSE), The Finnish Chapter of INCOSE, retrieved 11/22/2012.
- [336] Stéphane S. Somé. Enhancement of a Use Cases Based Requirements Engineering Approach with Scenarios. In *12th Asia-Pacific Software Engineering Conference (APSEC '05)*, pages 25–32, Dec 2005.
- [337] Andreas Spillner, Tilo Linz, and Hans Schaefer. *Software Testing Foundations: A Study Guide for the Certified Tester Exam*. Rocky Nook, pages 105 et seq., 127 et seq., 139 et seq., 3. edition, 1 2011.
- [338] J. M. Spivey. An introduction to Z and formal specifications. *Software Engineering Journal*, 4(1):40–50, January 1989.

- [339] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer, pages 131-133, December 1973.
- [340] Herbert Stachowiak. *Modelle - Konstruktion der Wirklichkeit*. Wilhelm Fink Verlag, pages 17-86, 1983.
- [341] Mirosław Staron, Ludwik Kuzniarz, and Ludwik Wallin. Case Study on a Process of Industrial MDA Realization: Determinants of Effectiveness. *Nordic Journal of Computing*, 11(3):254–278, September 2004.
- [342] Gunter Stein. *Respect the Unstable - the practical, physical (and sometimes dangerous) consequences of control must be respected, and the underlying principles must be clearly and well taught*. Feature in IEEE Control Systems Magazine, pages 12-25, 2003.
- [343] John Strasburger. SAE ARP 4754A Linkage with DO-178 and DO-254. In *2011 National Software and Airborne Electronic Hardware Conference (FAA 2011 SW & AEH)*, St. Louis, Missouri, USA, September 13-15 2011. presented on the 14th September.
- [344] Louis H. Sullivan. The Tall Office Building Artistically Considered. *Lippincott's Magazine*, 1896.
- [345] Alistair Sutcliffe. Scenario-based requirements analysis. *Requirements Engineering*, 3(1):48–65, 1998.
- [346] Paul A. Swatman. Using Formal Specification in the Acquisition of Information Systems: Educating Information Systems Professionals. *Proceedings of the Z User Workshop*, Springer Verlag, J.P. Bowen (Editor), J.E. Nicholls (Editor), pages 205–239, 14-15 December 1992.
- [347] Technology Systems Engineering Forum Office of the Under Secretary of Defense (Acquisition and Logistics) Defense Systems. Acquisition Modeling and Simulation Master Plan. http://www.acq.osd.mil/se/docs/AMSMP_041706_FINAL2.pdf, 2006. U.S. Department of Defense (DoD), retrieved 01/11/2013.
- [348] Stephan Thesing, Jean Souyris, Reinhold Heckmann, Famantanantsoa Randimbivololona, Marc Langenbach, Reinhard Wilhelm, and Christian Ferdinand. An abstract interpretation-based timing validation of hard real-time avionics software. *International Conference on Dependable Systems and Networks (DSN'03)*, pages 625–632, 22-25 June 2003.
- [349] Richard H. Theyer and Merlin Dorfman. *System and Software Requirements Engineering*. IEEE Computer Society Press, U.S., 1994.
- [350] Patrick Thibodeau. Healthcare.gov website 'didn't have a chance in hell'. <http://www.computerworld.com/article/2486426/healthcare-it/healthcare-gov-website--didnt-t-have-a-chance-in-hell-.html>, 2013. Interview with Jim Johnson, The Standish Group, Article Oct 21, Computerworld, International Data Group.
- [351] Donald Thomas and Philip Moorby. *The Verilog Hardware Description Language*. Springer US, 5th ed. 2002, 2nd printing 27th October 2008, 2008.

- [352] Charles Cyril Turner. *The Romance of Aeronautics: An Interesting Account of the Growth & Achievements of All Kinds of Aerial Craft*. Philadelphia, J.P. Lippincott Company, chapter VII: Lilienthal and Pilcher, page 87, 1912.
- [353] Michael L. Ulrey, Daniel L. Palumbo, and David M. Nicol. Case Study: Safety Analysis of the NASA/Boeing Fly-by-Light Airplane Using a New Reliability Tool. *Proceedings of the Annual Reliability and Maintainability Symposium, International Symposium on Product Quality and Integrity*, pages 318 – 325, 22-25 January 1996.
- [354] Koordinierungs und Beratungsstelle der Bundesregierung für Informationstechnik in der Bundesverwaltung (KBSt). The V-Model XT Version 1.3. <http://v-modell.iabg.de/dmdocuments/V-Modell-XT-Gesamt-Englisch-V1.3.pdf>, 2009. copyright: Federal Republic of Germany, retrieved 11/15/2012.
- [355] Mark Utting and Bruno Legeard. *Practical Model-Based Testing. A Tools Approach*. Morgan Kaufmann, 2007.
- [356] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.
- [357] Parham Vasaiely. *Model-Based Design, Verification and Validation of Systems using SysML and Modelica*. Master Thesis, Faculty of Engineering and Computer Science, Hamburg University of Applied Sciences, 2011.
- [358] Marc Virilli and Joerg Reitmann. Performance monitoring of In-Flight Entertainment systems - Airbus vision. *FAST 39 - Flight Airworthiness Support Technology*), 39:2–11, December 2006. available online, retrieved 07/11/2014.
- [359] Kerry Wagner, Brockwell, Daniels, Loesh, and Gosnell. Use of a Model-Based Approach to Minimize System Development Risk and Time-to-Field for New Systems. *13th Annual Systems Engineering Conference*, 25 - 28 October 2010.
- [360] Robert Wall and Pierre Sparaco. Cabin Control. *Aviation Week & Space Technology*, 162(14):46, 2005.
- [361] James Wallace. Hard decisions ahead for Boeing. <http://www.seattlepi.com/business/article/Hard-decision-ahead-for-Boeing-1100309.php>, 2002. Seattle Post-Intelligencer (Seattle-PI), retrieved 11/16/2012.
- [362] James Wallace. Airbus / Lufthansa information quoted online by James Wallace. <http://blog.seattlepi.com/aerospace/2007/03/29/a380-ride-and-factoids/>, 2011. Seattle Post-Intelligencer (Seattle-PI), retrieved 11/16/2012.
- [363] Fernando Wanderley, Denis Silva da Silveira, Joao Araujo, and Ana Moreira. Transforming Creative Requirements into Conceptual Models. In *IEEE Seventh International Conference on Research Challenges in Information Science (RCIS 2013)*, pages 1–10, May 2013.
- [364] Thomas Weigert, Frank Weil, Kevin Marth, Paul Baker, Clive Jervis, Paul Dietz, Yexuan Gui, Aswin van den Berg, Kim Fleer, David Nelson, Michael

- Wells, and Brian Mastenbrook. *Experiences in deploying model-driven engineering*. Springer-Verlag Berlin, Lecture Notes in Computer Science, Volume 4745/2007, pages 35-53, 2007.
- [365] Tim Weilkens. *Systems Engineering mit SysML/UML: Modellierung, Analyse, Design*. dpunkt Verlag, pages 2 et seq., 33 et seq., 90 et seq., 352 et seq., 372 et seq., 2008.
- [366] Eric Weiner. New Boeing Airliner Shaped by the Airlines. <http://www.nytimes.com/1990/12/19/business/new-boeing-airliner-shaped-by-the-airlines.html?pagewanted=2&src=pm>, 1990. The New York Times, (December 19, 1990), retrieved 01/14/2013.
- [367] Michael Whalen, Steven P. Miller, and Darren Cofer. Formal Verification of Avionics Software in a Model-Based Development Process. *First International Workshop on Aerospace Software Engineering (AeroSE 07)*, May 21-22 2007.
- [368] Alexander Wichmann, Sven Jäger, Tino Jungebloud, Ralph Maschotta, and Armin Zimmermann. System Architecture Optimization With Runtime Re-configuration of Simulation Models. In *9th Annual IEEE International Systems Conference (SysCon 2015)*, pages 660–667, Vancouver, Canada, April 2015.
- [369] Jonathan Wickert and Kemper E. Lewis. *An Introduction to Mechanical Engineering*. CENGAGE Learning, 2012.
- [370] Karl E. Wiegers. *Software Requirements, Second Edition - Practical techniques for gathering and managing requirements throughout the product development lifecycle*. Microsoft Press, 2003.
- [371] Mark Wiegmann. Vorlesung Elektronische Kabinensysteme (EKS). lecture notes, 2011. summer term lecture, Hamburg University of Applied Sciences.
- [372] Mark Wiegmann. Electronic cabin systems, data communication for aircraft. lecture notes, 2012. lecture at: Hamburg University of Applied Sciences, 25.05.2012.
- [373] Mark Wiegmann. Analyse und Synthese eines grundlegenden Kabinensystems mit Hilfe modellbasierter Entwicklungsmethoden: SIMKAB B.2 (LuFo IV, 2nd Call). final project report (public), Reportnr. / Förderkennzeichen: 20K0805R, Verbund-Nr. 01066673, 2014. Technische Informationsbibliothek und Universitätsbibliothek Hannover (TIB/UB).
- [374] Mark Wiegmann. Elektronik in der Flugzeugkabine - Studienvertiefung Kabine und Kabinensysteme. lecture notes, 2014. originally presented at "WednesdayTI", Series of public lectures of the Faculty of Engineering and Computer Science at Hamburg University of Applied Sciences, 25.05.2012.
- [375] Michael Wooldridge. Intelligent Agents: The Key Concepts. In *Multi-Agent-Systems and Applications II, 9th ECCAI-ACAI/EASSS 2001, AEMAS 2001, HoloMAS 2001, Selected Revised Papers*, pages 3–43, 2001.

- [376] Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: theory and practice. *The Knowledge Engineering Review*, 10:115–152, 6 1995.
- [377] Wilbur Wright. *Otto Lilienthal*. published posthumously in: Aero Club of America Bulletin, 1912.
- [378] Wilbur Wright and Marvin W. McFarland (ed.). *The Papers of Wilbur and Orville Wright, Including the Chanute-Wright Letters and Other Papers of Octave Chanute. Vol 1: 1899-1905. Vol. 2: 1906-1948*. McGraw Hill Higher Education, 2000.
- [379] Ingo Wuggetzer, Axel Becker, and Nicolas Tschechne. Passenger at Heart, Airline in Mind. *VDI Mensch & Technik*, 17:12–13, 2011.
- [380] Heinz-Dietrich Wuttke and Karsten Henke. *Schaltsysteme: Eine automatenorientierte Einführung (Pearson Studium - IT)*, chapter 5, pages 190–197. Pearson Studium, 1 edition, 12 2002.
- [381] Ghim-Lay Yeo. Getting Personal. *Flight International*, 181:33–37, 20-26 March 2012.
- [382] Jia Yu and Rajkumar Buyya. A Taxonomy of Scientific Workflow Systems for Grid Computing. *SIGMOD Rec.*, 34(3):44–49, September 2005.
- [383] William Yurcik. Simulation. <http://www.encyclopedia.com/doc/1G2-3401200119.html>, 2004. Computer Sciences from Encyclopedia.com, retrieved 10/07/2014.
- [384] Joachim Zarrat. *Qualitätsmanagement nach DIN EN 9100:2010 in der Luftfahrt, Raumfahrt und Verteidigung: Wegweiser für die praktische Umsetzung*. Beuth Verlag, 2012.
- [385] Bernard P. Zeigler, Tag Gon Kim, and Herbert Praehofer. *Theory of Modeling and Simulation*. Academic Press, Inc., Orlando, FL, USA, 2nd edition, 2000.
- [386] Armin Zimmermann. *Stochastic Discrete Event Systems: Modeling, Evaluation, Applications*. Springer Berlin Heidelberg, softcover reprint of hardcover 1st ed. 2008 edition, 2009.
- [387] Jens Zimmermann and Sascha Kirstan. Auszug aus der Studie Kosten und Nutzen der modellbasierten Entwicklung eingebetteter Softwaresysteme im Automobil. http://www.altran.de/fileadmin/medias/DE.altran.de/documents/Studien/auszug_studie_mbse.pdf, 2010. TU-München and ALTRAN Technologies, retrieved 01/23/2013.
- [388] Daniel Zinn. *Modeling and Optimization of Scientific Workflows*. PhD thesis, University of California at Davis, 2010. Ph. D. Dissertation.
- [389] Konrad Zuse. *Der Computer - Mein Lebenswerk: 100 Jahre Zuse*. Springer Berlin Heidelberg, 2010.

A. MR-MBSE Library

A.1 Top-Level Library

Date:	Di 9. Feb 20:53:07 2016
Version:	0.0 10/23/2015
Author:	Nils Fischer

Top-level MLDesigner library for Minimum Risk Model-based Systems Engineering (MR-MBSE). Includes Executable Workflows / Simulation Sets libraries, Mission Model libraries, Service Model libraries, System and System Environment Configuration Model libraries, Customization Model libraries and Validation Report libraries. Moreover, this library contains examples for MR-MBSE development of a basic Cabin Management System (CMS) and Avionics Optimization.

Current Location: MLD_USER/Thesis_NilsFischer

Import Models:



SystemDS

Composite Data Structures:

Root.ACTOR	Used for communication between actor modules.			
AMIS_Label	Root.String		aLabel	Determines a specific Atomic Mission Model (AMIS) that is associated with the respective actor.
NAME	Root.String		aName	Specific name of the actor. This is essential to map mission actors and system actors.
ID	Root.Integer	(0, Inf)	1	ID of the actor.
TYPE	Root.ENUM.ACTOR_type		HUMAN	Specifies the basic type of an actor.
ROLE	Root.ENUM.Actor_State		ACTIVE	Specifies the basic role of an actor.
Signal	Root			Carries the information / data that is to be exchanged during simulation (data flow).
PING	Root.ENUM.YesNo		No	Used for actor model initialization.
GetCurrentState	Root.ENUM.Boolean		FALSE	Used to request the last valid data of a system actor module.
Root.ACTOR_Customization	Used to exchange customization model data.			
Name	Root.String		aName	Specific name of the actor. This is essential to map mission actors and system actors.
ID	Root.Integer		0	ID of the actor.
Configuration	Root			Carries the information / data that is to be exchanged during simulation (customization exchange).
Root.ID_Range	Used to specify a range of model component IDs, e.g. for aggregated actor modules.			
Start_ID	Root.Integer	(0, Inf)	1	Start of the ID range.
Final_ID	Root.Integer	(0, Inf)	1	End of the ID range.
Root.Objective	Used to determine quality objective or non-function parameter ranges.			
Objective_Name	Root.String		aName	Unique name of the objective.
Lower_Bound	Root.Float		0	Lower objective boundary.
Mean	Root.Float		0	Mean objective value or target value.
Upper_Bound	Root.Float		0	Upper objective boundary.
WeightFactor	Root.Float	[0, 1]	0	Objective weight factor (range 0-1).
Root.Quality_Mission_Dynamic	Used for communication of quality objective nodes (dynamic objectives).			

Element_Name	Root.String		aName	Determines the name of the ES module providing the information.
ID	Root.Integer		0	Element ID.
Parameter_Value	Root.Float		0	Used to store a non-functional element parameter value.
Root.Quality_Mission_Static Used for communication of quality objective nodes (static objectives).				
Element_Name	Root.String		aName	Determines the name of the ES module providing the information.
ID	Root.Integer		0	Element ID.
Objective	<u>Root.Objective</u>			Used to store a non-functional element parameter value in form of an objective.
Root.Sys_Env_Configuration Used for system and system environment configuration model information exchange.				
Temporal_Address	Root.Integer	(0, Inf)	123	Temporary memory address for configuration data.
Configuration	Root			Configuration Information.
Root.ValidationDS Validation information used for validation report generation e.g. for online and post simulation analysis.				
System_Name	Root.String		aName	Determines a specific system or subsystem name.
System_ID	Root.Integer	[0, Inf)	0	
Mission_Label	Root.String		aName	Determines a specific mission label.
Mission_ID	Root.Integer	[0, Inf)	0	Determines a specific mission identifier.
Atomic_Mission_Label	Root.String		aName	Determines a specific atomic mission label.
Atomic_Mission_ID	Root.Integer	[0, Inf)	0	Determines a specific atomic mission identifier.
Quality_Objective_Parameter	Root.String		aName	Determines a specific quality objective parameter.
Service_Label	Root.String		aName	Determines a specific service model label.
Service_ID	Root.Integer	[0, Inf)	0	Determines a specific service model identifier.
Scenario_Label	Root.String		nA	Determines a specific scenario model label.
Scenario_ID	Root.Integer		0	Determines a specific scenario model identifier.
Requirement_Type	<u>Root.ENUM.Requirement_Type</u>		Functional	Determines the type of requirement.
Validation_Status	<u>Root.ENUM.Validation_Status</u>		Validated	Stores the validation status of an associated model.
Information	Root.String		anyInfo	Any additional information.
Quality_Objectives	<u>Root.Vector.Objectives_List</u>			(vector) Used to store a set of quality objectives.

Enumeration Data Structures:

<u>Root.ENUM.Actor_State</u>	Determines a basic actor behavior class.
ACTIVE	0
PASSIVE	1
MIXED	2
<u>Root.ENUM.ACTOR_type</u>	Determines a basic actor type.
HUMAN	0
SUBSYSTEM	1
COMPONENT	2
ENVIRONMENT	3
<u>Root.ENUM.Requirement_Type</u>	Determines the type of an requirement.
Functional	0
Non_Functional	1
Mixed	2
<u>Root.ENUM.Validation_Status</u>	Determines two possible validation states.
Validated	0
Failed	1

Vector Data Structures:

<u>Root.Vector.Objectives_List</u>	Used to store a set of objectives.
Vector Type:	<u>Root.Objective</u>

Models:

 <u>AvionicsOPT</u>	Library for the optimization of avionics system architectures. Derived from diploma thesis by Nils Fischer, TU Ilmenau 2007.
 <u>CMS</u>	Library for MR-MBSE development of a basic Cabin Management System (CMS).
 <u>Mission_Model_Elements</u>	Library with basic modules for mission model development.
 <u>Report_Generation</u>	Library with basic modules for simulation report generation.
 <u>Service_Model_Elements</u>	Library with basic modules for service model development.
 <u>Executable_Workflow_Elements</u>	Basic library for creating Executable Workflows and Simulation Sets, e.g. for Mission Models.

A.2 Simulation Set Library

Date:	Di 9. Feb 20:53:07 2016
Version:	0.0 10/23/2015
Author:	Nils Fischer

Minimum Risk Model-based Systems Engineering (MR-MBSE) library for creating Executable Workflows and Simulation Sets, e.g. for Mission Models.

Current Location: MLD_USER/Thesis_NilsFischer/Workflow_Elements

Models:

	Branch_End_Node	Basic branch end node. Used to terminate a specific branch of the overall control flow.
	Decision	Basic true/false decision node. Used to determine a condition within a control flow.
	Decision_wElse	Basic decision node with two conditions and else branch. Used to determine a fork with multiple conditions within a control flow.
	Finish	Basic finish node. Used to terminate a control flow.
	Global_Finish_plain	Basic global finish node. Used to terminate a top-level control flow.
	Global_Start	Basic global start node. Used to start a top-level control flow.
	Initialize	Basic start node. Used to start a control flow.
	Join_Synchronized_H	Basic horizontal synchronized join node with two inputs. Used to synchronize two events before executing a subsequent control flow node.
	Join_Synchronized_H_3	Basic horizontal synchronized join node with three inputs. Used to synchronize three events before executing a subsequent control flow node.
	Join_Synchronized_H_4	Basic horizontal synchronized join node with four inputs. Used to synchronize four events before executing a subsequent control flow node.
	Join_Synchronized_V	Basic vertical synchronized join node with two inputs. Used to synchronize two events before executing a subsequent control flow node.
	Join_Synchronized_V_3	Basic vertical synchronized join node with three inputs. Used to synchronize three events before executing a subsequent control flow node.
	Join_Synchronized_V_4	Basic vertical synchronized join node with four inputs. Used to synchronize four events before executing a subsequent control flow node.
	Join_Unsynchronized_multi	Basic unsynchronized join node with arbitrary number of inputs. Used to join the control flow with subsequent control flow node execution.
	Process_Handling	Basic library for execution and control of external simulations (PTCL / C++)
	Signal_Customization_Send	Basic signal send node for customization information. Used to transmit customization model data between connected nodes.
	Signal_Receive	Basic signal receive node for any information. Used to exchange data between connected nodes.
	Signal_Receive_new	Basic signal receive node for any information (alternative version). Used to exchange data between connected nodes.
	Signal_Send	Basic signal send node for any information. Used to exchange data between connected nodes.
	Simulation_Set_Elements	Predefined module for external simulation (PTCL / C++) execution and control (combined model).
	Split_H_multi	Basic horizontal synchronized split node with arbitrary number of inputs. Used to synchronously trigger subsequent control flow nodes.
	Split_V_multi	Basic vertical synchronized split node with arbitrary number of inputs. Used to synchronously trigger subsequent control flow nodes.
	Timed_Event	This module triggers a timed event. It is linked to a start condition event (part of the init node or an activity node). When the start condition is fulfilled, the specified amount of time needs to pass before the event trigger is activated.
	Timed_Event_Randomize	This module triggers a timed event. It is linked to a start condition event (part of the init node or an activity node). When the start condition is fulfilled, the specified amount of time needs to pass before the event trigger is activated.

Simulation Set Elements library

Predefined module for external simulation (PTCL / C++) execution and control (combined model).

Current Location: MLD_USER/Thesis_NilsFischer/Workflow_Elements/Simulation_Set_Elements

Models:

	Executable_Simulation_Object_Node	Used to execute and control an external simulation. An external simulation with specified path is executed and monitored. Upon completion of the external simulation, a control flow token is provided at the output port.
---	---	--

Process Handling library

Basic library for execution and control of external simulations (PTCL / C++)

Date:	So 7. Feb 20:31:51 2016
Version:	0.0 12/14/2014
Author:	Tino Jungebloud

Current Location: MLD_USER/Thesis_NilsFischer/Workflow_Elements/Process_Handling

Models:

	<u>Process_Base</u>	Base primitive for external process handling.
	<u>Process_GetState</u>	Primitive to monitor external processes.
	<u>Process_Kill</u>	Primitive to force the end of an external process (forced termination).
	<u>Process_Start</u>	Primitive to trigger/start an external process (system call).

A.3 Mission Model Library

Date:	Di 9. Feb 20:53:07 2016
Version:	0.0 10/23/2015
Author:	Nils Fischer

Library with basic modules for mission model development.

Current Location: MLD_USER/Thesis_NilsFischer/Mission_Model_Elements

Models:

	Atomic_Mission_Models	Library with basic modules for atomic mission model development.
	Envrionment_Configuration_Models	Library with basic modules for environment configuration model development.
	Quality_Objective_Models	Library with basic modules for quality objective model development.

Atomic Mission Models library

Library with basic modules for atomic mission model development.

Current Location: MLD_USER/Thesis_NilsFischer/Mission_Model_Elements/Atomic_Mission_Models

Models:

	AMIS_Actors	Library with basic actor modules for atomic mission model development.
	ES_Actors	Library with basic actor modules for executable overall system specification development.
	Std_Atomic_Mission_Module	Pre-defined, empty mission module template.
	Std_Continuous_Atomic_Mission_Module	Pre-defined, empty mission module (continuous) template.
	Tools	Support library for atomic mission model development (misc).

AMIS Actors library

Library with basic actor modules for atomic mission model development.

Current Location: MLD_USER/Thesis_NilsFischer/Mission_Model_Elements/Atomic_Mission_Models/AMIS_Actors

Models:

	Mission_Actor	Generic actor module (single actor) for executable requirements specification / mission model development.
	Mission_Actor_Actuator	Actuator actor module (task-specific actor) for executable requirements specification / mission model development.
	Mission_Actor_Environment	Environment actor module (task-specific actor) for executable requirements specification / mission model development.
	Mission_Actor_Sensor	Sensor actor module (task-specific actor) for executable requirements specification / mission model development.
	Mission_Actor_Subsystem	Subsystem actor module (task-specific actor) for executable requirements specification / mission model development.
	Mission_Customization_Actor	Customization actor module (customization-specific actor) for executable requirements specification / mission model development.

ES Actors library

Library with basic actor modules for executable overall system specification development.

Current Location: MLD_USER/Thesis_NilsFischer/Mission_Model_Elements/Atomic_Mission_Models/ES_Actors

Models:

	System_Actor_aggregated	Generic actor module (aggregated actor) for executable overall system specification development.
	System_Actor_single	Generic actor module (single actor) for executable overall system specification development.
	System_Actor_single_Actuator	Actuator actor module (task-specific actor) for executable overall system specification development.
	System_Actor_single_Environment	Environment actor module (task-specific actor) for executable overall system specification development.
	System_Actor_single_Sensor	Sensor actor module (task-specific actor) for executable overall system specification development.

	System_Actor_single_Subsystem	Subsystem actor module (task-specific actor) for executable overall system specification development.
	System_Mission_Customization_Actor	Customization actor module (customization-specific actor) for executable overall system specification development.

Atomic Mission Models Tools library

Support library for atomic mission model development (misc).

Current Location: MLD_USER/Thesis_NilsFischer/Mission_Model_Elements/Atomic_Mission_Models/
Tools

Models:

	AMISActor_Receiver	AMIS actor message receiver port.
	Change_Parameter_Values	Used to change actor parameters.
	End_Mission	Ends current mission.
	ESActor_Receiver	ES actor message receiver port.
	ESActor_Receiver_multi	ES-multi Actor message receiver port.
	ESActor_Sender_multi	ES-multi Actor message sender port.
	ESCustomActor_Receiver	ES-customization Actor message receiver port.
	InitialCustomActorCheckup	ES-customization actor initial configuration check.
	InitialESActorCheckup	ES actor initial configuration check.
	SendCustomToESModel	Custom DS to ES model sender primitive.
	SendToAMISModel	DS to AMIS model sender primitive.
	SendToESModel	DS to ES model sender primitive.
	Start_Mission	Start mission primitive.

Environment Configuration Models library

Library with basic modules for environment configuration model development.

Current Location: MLD_USER/Thesis_NilsFischer/Mission_Model_Elements/Environment_Configuration_Models

Models:

	Configure_System_or_Environment	Used as part of mission models in order to relay configuration information to executable specification models.
	Configure_System_or_Environment_from_File	Reads System or Environment Configurations from a file (line by line) and sends it to an executable specification model each time when triggered.
	Handle_System_or_Environment_Configuration	Used by customized System and Environment configuration change modules, primitives or FSMs in order to relay configuration information to executable specification models.
	System_or_Environment_Configuration	Used by executable specification models. Configuration memory is linked to an external memory that represents a system or environment specific configuration data base (DB). This configuration may be changed by mission model entities in order to change mission specific parameters.
	tools	Support library for atomic mission model development (misc).

Environment Configuration tools library

Support library for atomic mission model development (misc).

Current Location: MLD_USER/Thesis_NilsFischer/Mission_Model_Elements/Environment_Configuration_Models/tools

Models:

	Produce_Data_from_File	Load environment configuration from file.
	Produce_Data_from_File_Triggered	Load environment configuration from file when triggered.
	random_goto	Random address management.
	UpdateCFG	Configuration update primitive.

Quality Objective Models library

Library with basic modules for quality objective model development.

Current Location: MLD_USER/Thesis_NilsFischer/Mission_Model_Elements/Quality_Objective_Models

Models:

	NF_Observer	Basic non-functional parameter observer module for executable overall system specification model development.
	Quality_Mission_Model_4VirtualPrototype	Basic quality mission / non-functional parameter module for virtual prototype development.
	Quality_Mission_Model_continuous_4VP_standalone	Basic quality mission / non-functional parameter module of type continuous for virtual prototype development.
	Quality_Mission_Model_continuous_detached	Basic quality mission / non-functional parameter module of type continuous for executable overall system specification model development.
	Quality_Mission_Model_continuous_triggered	Basic quality mission / non-functional parameter module of type continuous for executable overall system specification model development. Needs to be triggered first before execution.
	Quality_Mission_Model_static	Basic quality mission / non-functional parameter module of type static for executable overall system specification model development.
	Quality_Mission_Model_static_4VP_standalone	Basic quality mission / non-functional parameter module of type static for virtual prototype development.
	tools	Support library for atomic mission model development (misc).

Quality Objective tools library

Support library for atomic mission model development (misc).

Current Location: MLD_USER/Thesis_NilsFischer/Mission_Model_Elements/Quality_Objective_Models/tools

Models:

	ShowParameterUncertainty_FLOAT	Show float type quality parameter with boundaries after simulation.
	ShowParameterUncertainty_wObjective	Show objective type quality parameter with boundaries after simulation.

A.4 Service Model Library

Date:	Di 9. Feb 20:53:07 2016
Version:	0.0 10/23/2015
Author:	Nils Fischer

Library with basic modules for service model development.

Current Location: MLD_USER/Thesis_NilsFischer/Service_Model_Elements

Models:

	Std_Continuous_Service_Model_Structure	Pre-defined, empty service module (continuous) template.
	Std_Encapsulated_Scenario	Pre-defined, empty service module template.
	tools	Support library for service model development (misc).

Service_Model_Elements tools library

Support library for service model development (misc).

Current Location: MLD_USER/Thesis_NilsFischer/Service_Model_Elements/tools

Models:

	Scenario_Port_SendReceive	Basic module to receive scenario information from atomic mission models.
	Scenario_Port_SendReceive_Mem	Basic module to receive scenario information from atomic mission models (with attached memory).

A.5 Validation Report Library

Date:	Di 9. Feb 20:53:07 2016
Version:	0.0 10/23/2015
Author:	Nils Fischer

Library with basic modules for simulation report generation.

Current Location: MLD_USER/Thesis_NilsFischer/Report_Generation

Models:

	Basic_Components	Basic file generation tools.
	Validation_Elements	Library with basic modules for automated and interactive validation report generation.

Basic Components library

Basic file generation tools.

Current Location: MLD_USER/Thesis_NilsFischer/Report_Generation/Basic_Components

Models:

	Create_Basic_CSV_File	Creates the basic structure of a requirements specification file with title, issue and date.
	Create_Basic_Requirement_File	Creates the basic structure of a requirements specification file with title, issue and date.
	Produce_Data_from_Directory	Load data from input file.
	Set_Line_of_File	Determine specific line of a file to write.

Validation Elements library

Library with basic modules for automated and interactive validation report generation.

Current Location: MLD_USER/Thesis_NilsFischer/Report_Generation/Validation_Elements

Models:

	Generate_System_Validation_Report	Basic module for creation of a global automated validation report.
	Generate_System_Validation_Report_CSV	Basic module for creation of a global automated validation report (CSV type) with tabular structure.
	Generate_System_Validation_Report_wEndCtrl	Basic module for creation of a global automated validation report with alternative control flow.
	Interactive_Validation	Library with basic modules for interactive validation report generation.
	Report_Validation_Info	Basic module for validation information of executable overall system specification models (validation probe).
	Report_Validation_Info_Scenario	Basic module for validation information of scenario models (validation probe).
	tools	Support library for model validation (misc).

Validation Elements tools library

Support library for model validation (misc).

Current Location: MLD_USER/Thesis_NilsFischer/Report_Generation/Validation_Elements/tools

Models:

	DS_member_name_extractor	Get specific member name of a data structure.
	DS_member_values_extractor	Get specific member value of a data structure.
	GetStringfromValue	Extract info from a value and convert to string.
	Quicksort_DS_Numerical	Performs a Quicksort algorithm for incoming data structures (DS_In) and corresponding index values received at Integer_IN). The algorithm is activated by triggering input
	SendGlobalwAck	Global validation file pointer management.
	String2File_wTab	Writes a string to a csv file.

B. Basic Cabin Management System

In this chapter, major functions, components and parameter settings are provided for the cabin management system (CMS) example elaborated in chapter 5. Firstly, components of the developed executable requirements specification are provided, including an overview of developed atomic mission models and a full description for the developed scenario flowchart of the atomic mission model *PAX Call* in chapter 5.4.5. Secondly, the overall data and customization model of the developed CMS is provided. Thirdly, the two automatically generated CMS validation reports are provided.

B.1 Developed Atomic Mission Models

As part of the mission model example provided in chapter 5.4.1, seven different atomic mission (AMIS) models with associated services were developed. Each AMIS will be described in the following paragraphs, including a description of why each model was regarded to be essential and was thus included in the overall design.

Cabin to Ground Service Intercommunication

Identifier	Cabin-Ground Service Intercom
Actors	Human actor <i>GRND_2_CAB_Caller</i> represents airport ground staff while human actor <i>CAB_2_GRND_Caller</i> represents cabin attendants.
Purpose	During aircraft operation on ground, it is necessary to provide intercommunication capabilities for information exchange between airport ground staff and cabin crew. Typical communication topics include potable water and waste tank management and aircraft luggage management. Ground staff members may directly establish a connection to cabin crew members e.g. via a ground service plug interaction point (IP) while cabin attendants may use handset IPs.

Included Service(s)	One service model is provided. This model describes the general use case intended for this AMIS model, i.e. a two-way communication between <i>CAB_2_GRND_Caller</i> and <i>GRND_2_CAB_Caller</i> .
Trigger	<i>CAB_2_GRND_Caller</i> initiates new service intercommunication call.
Precondition(s)	The aircraft is required to be on ground with power generation systems active and no service intercommunication currently active.
Expected Result	<i>CAB_2_GRND_Caller</i> and <i>GRND_2_CAB_Caller</i> have successfully performed an aircraft cabin to ground service communication call.
Scenario	1) <i>CAB_2_GRND_Caller</i> initiates a new ground service call. 2) Next, human machine interface (HMI) feedback is expected at cabin and ground staff IPs, including visual and audible information. 3) After a short waiting period (configurable length), <i>GRND_2_CAB_Caller</i> accepts the incoming call. 4) Subsequently, both parties exchange audio information (service call). For each source audio information, there must be a corresponding reception of information at the associated audio sink. 5) At the end of communication, <i>CAB_2_GRND_Caller</i> terminates the currently active call and 6) HMIs for active call indications are expected to indicate no remaining call activities.
Scenario Variation(s)	n/a
Quality Constraint(s)	Visual indications (indication light properties) and audio qualities (volume level) are monitored during scenario execution.
Postcondition(s)	No service intercommunication shall be active.
Why Essential?	Covered by most systems on the market, crucial service feature for ground operations (stakeholder requirement), direct link to a major entity of system environment (ground services)

Inter-Aircraft Communication

Identifier	Cabin-Cabin Intercom / Cabin-Cockpit Intercom
Actors	Human actor <i>CAB_2_CAB_Caller</i> represents cabin attendants that are involved in cabin to cabin communication (without cockpit intercommunication).

Purpose	During aircraft operation on ground or in flight, it is necessary to provide intercommunication capabilities for information exchange between cabin crew members and between cabin crew members and cockpit crew. Typical communication topics include on-board service requests or in-flight information. Aircraft crew members may establish a connection to other crew members via a two-way communication IP e.g. via handset IPs.
Included Service(s)	One service model is provided. This model describes the general use case intended for this AMIS model, i.e. a two-way communication between two entities of actor <i>CAB_2_CAB_Caller</i> .
Trigger	One entity of actor <i>CAB_2_CAB_Caller</i> initiates a new cabin crew to cabin crew intercommunication call.
Precondition(s)	The aircraft is required to be powered with no currently active intercommunication between the intended set of actors at specific positions within the aircraft.
Expected Result	Two entities of actor <i>CAB_2_CAB_Caller</i> have successfully performed a cabin crew to cabin crew intercommunication call.
Scenario	1) One entity of actor <i>CAB_2_CAB_Caller</i> initiates a new cabin call. 2) Next, HMI feedback is expected at cabin crew IPs, including visual and audible information. 3) After a short waiting period, another entity of actor <i>CAB_2_CAB_Caller</i> accepts the incoming call and 4) both parties exchange audio information. For each source audio information, there must be a corresponding reception of information at the associated audio sink. 5) At the end of communication, entity one of actor <i>CAB_2_CAB_Caller</i> terminates the currently active call and 6) HMIs for active call indications are expected to indicate no remaining call activities.
Scenario Variation(s)	n/a
Quality Constraint(s)	Visual indications (indication light properties and indication display properties), audible indications (cabin chime properties) and audio qualities (volume level) are monitored during scenario execution.
Postcondition(s)	No intercommunication between both actor entities shall be active.
Why Essential?	Covered by most systems on the market, crucial service feature for cabin operations (stakeholder requirement), required for aircraft safety/security (legal requirement)

Passenger Address

Identifier	Public Address / Passenger Address / PA
Actors	Human actor <i>CAB_PA</i> represents aircraft crew members while human actor <i>PAX_PA</i> represents a set of passengers at different aircraft zones, seat rows and seats.
Purpose	During aircraft operation on ground or in flight, it is necessary to be able to provide public address (PA) capabilities for passenger information. This information is provided by cabin or cockpit crew members. Typical passenger information topics include safety information, on-board service information or in-flight information. Aircraft crew members may establish a public address to all people aboard an aircraft or to selected areas by using a one-way communication IP e.g. a handset IP.
Included Service(s)	One service model is provided. This model describes the general use case intended for this AMIS model, i.e. a one-way communication between actors <i>CAB_PA</i> and <i>PAX_PA</i> . Moreover, the service mode provided includes a number of 1677 use case variations. Variations include direct PAs and PAs to all available aircraft zones, priority variations as well as objective variations for audio frequency and PA duration as well as combinations thereof.
Trigger	<i>CAB_PA</i> initiates a new PA to one or more aircraft zones with specific priority and audio quality settings.
Precondition(s)	The aircraft is required to be powered (essential or non-essential power provision) with no currently active PA.
Expected Result	<i>CAB_PA</i> has successfully performed a PA for a set of specific zones with specific priority and corresponding audio properties.
Scenario	1) A PA source actor of type <i>CAB_PA</i> initiates a new PA call with specific priority and audio configuration for the entire aircraft, a single zone or a set of zones. 2) Next, visual HMI feedback is expected at cabin crew IPs. 3) If successful, <i>CAB_PA</i> provides audio to perform the intended PA with specific duration. 4) Successful reception of PA audio is monitored by entities of <i>PAX_PA</i> at respective audio sinks. 5) After that, the PA is ended by <i>CAB_PA</i> and 6) HMI feedback is observed at expected cabin crew and passenger IPs.
Scenario Variation(s)	After each scenario run, scenario parameters for intended PA zone, priority and audio quality characteristics are changed before the scenario is executed again. If all sets of intended variations have been performed successfully, the overall scenario is ended.

Quality Constraint(s)	PA audio quality criteria (audio frequency, duration) and audio sink characteristics (zone, priority) are monitored during scenario execution.
Postcondition(s)	The currently executed PA must have ended.
Why Essential?	Covered by most systems on the market, crucial service feature for cabin operations (stakeholder requirement), required for aircraft safety/security (legal requirement)

Passenger Calls for Assistance

Identifier	Passenger Call / PAX Call
Actors	Human actor <i>PAX_Caller</i> represents passengers that require assistance. Human actors <i>PAX_Call_Visual</i> and <i>PAX_Call_Audible</i> represent cabin attendants that will provide assistance while customization actor <i>PAX_Call_Customization</i> is used for customization.
Purpose	During aircraft operation on ground or in flight, it is necessary to provide passengers with the ability to call cabin attendants for assistance. Typical assistance topics include on-board service, health assistance or in-flight information. Passengers can initiate and reset a call via a passenger-related IP e.g. via overhead consoles or handsets. Sometimes, PAX call IPs are also used by cabin attendants for service signaling, e.g. to get the attention of other cabin attendants during in-flight service (alternative usage of this AMIS).
Included Service(s)	One service model is provided. This model describes the general use case intended for this AMIS model, i.e. a call for assistance by <i>PAX_Caller</i> . Moreover, the service mode includes a number of 108 use case variations. Variations include PAX calls for each passenger as well as customization variations for indication light colors.
Trigger	One entity of actor <i>PAX_Caller</i> initiates a new PAX Call.
Precondition(s)	The aircraft is required to be powered with no currently active PAX Call between the intended set of actors at specific positions within the aircraft.
Expected Result	One entity of actor <i>PAX_Caller</i> has successfully performed a PAX Call and was assisted by one or more cabin attendants (<i>PAX_Call_Visual</i> or <i>PAX_Call_Audible</i>).

Scenario	1) One entity of actor <i>PAX_Caller</i> initiates a new PAX Call. 2) Next, HMI feedback is expected at cabin crew IPs, including visual and audible information. 3) After a short waiting period (configurable length or random bounded interval), 4) <i>PAX_Caller</i> receives assistance. 5) After <i>PAX_Caller</i> has been assisted, <i>PAX_Caller</i> terminates the currently active PAX Call (reset) and 6) HMIs for active call indications are expected to indicate no remaining PAX call activities for the respective actor.
Scenario Variation(s)	After each scenario run, scenario parameters and customization parameters are changed before the scenario is executed again. Scenario parameters include different PAX Caller entities while customization variations include visual indication properties. If all sets of intended variations have been performed successfully, the overall scenario is ended.
Quality Constraint(s)	Visual indications (indication light properties and indication display properties) and audio qualities (volume level) are monitored during scenario execution.
Postcondition(s)	No PAX call shall be active.
Why Essential?	Covered by most systems on the market, crucial service feature for cabin operations (stakeholder requirement), is the only system link between passengers and cabin crew

Lavatory Smoke Indication

Identifier	LAV Smoke
Actors	Environment actor <i>LAV_Smoke_Source</i> is part of the lavatory environment and is used for lavatory smoke state representation. Human actors <i>LAV_Smoke_Visual</i> and <i>LAV_Smoke_Audible</i> represent cabin attendants while customization actor <i>LAV_Smoke_Customization</i> is used for customization.
Purpose	During aircraft operation on ground or in flight, it is necessary to provide cabin attendants with the ability to monitor all cabin areas with regard to smoke in order to prevent or fight fires aboard an aircraft. Since lavatories may not be observed directly at all times (visual contact), a remote smoke indication is required.
Included Service(s)	Four service models are provided. Each of the service models includes a variation in audible smoke detection indication. While service model one uses no audible indication (cabin chime), all other three service models use different types of chimes for indication.
Trigger	One entity of actor <i>LAV_Smoke_Source</i> causes smoke within the associated lavatory.

Precondition(s)	There is no currently active lavatory smoke indication.
Expected Result	Smoke was caused within one lavatory, successfully indicated and treated by cabin attendants.
Scenario	1) One entity of actor <i>LAV_Smoke_Source</i> causes smoke within the associated lavatory. 2) Next, HMI feedback is expected at cabin crew IPs (<i>LAV_Smoke_Visual</i> and <i>LAV_Smoke_Audible</i>), including visual and audible information (if configured by customization). 3) After a short waiting period (configurable length or random bounded interval), 4) the source of LAV smoke is handled by cabin attendants. 5) If smoke has ceased to emanate, cabin HMIs are expected to indicate no active smoke indications.
Scenario Variation(s)	n/a
Quality Constraint(s)	Audio indication qualities (chime characteristics) are monitored during scenario execution.
Postcondition(s)	No lavatory smoke indication shall be active.
Why Essential?	Crucial service feature for cabin operations (stakeholder requirement), required for aircraft safety (legal requirement)

Passenger Notification System - Automated Fasten Seatbelt Indication

Identifier	PNS-FSB Signaling Auto / Return to Seat (RTS) Indication
Actors	Subsystem actors <i>LGSys</i> , <i>ENGsys</i> and <i>CPCS</i> are part of the cabin system environment and are used for subsystem state representations. Human actors of the cockpit environment <i>Cockpit_FSB_1</i> and <i>Cockpit_FSB_2</i> represent cockpit crew members while human actor <i>FSB_CAB</i> represents cabin crew members and <i>FSB_PAX</i> represents passengers.
Purpose	During aircraft operation on ground or in flight, it is necessary to provide fasten seatbelt (at seats) and return to seat indications (at cabin compartments e.g. lavatories) for passengers and cabin crew for safety reasons. An activated automated FSB-RTS indication provides information depending on the status of different aircraft subsystems.
Included Service(s)	One service model is provided. This model describes the general use case intended for this AMIS model, i.e. an automated FSB-RTS indication for cabin crew and passengers. In combination with different flight phases, this service is executed 16 times as part of the mission model described in chapter 5.4.1.
Trigger	One or more of the subsystem actors <i>LGSys</i> , <i>ENGsys</i> or <i>CPCS</i> change their state while PNS-FSB Signaling Auto is active.

Precondition(s)	Either actor <i>Cockpit_FSB_1</i> or <i>Cockpit_FSB_2</i> (customization dependent, logical actor assignment expression, LAAE) has activated automatic FSB-RTS indication.
Expected Result	FSB-RTS information is indicated correctly for each phase of aircraft operation.
Scenario	1) Subsystem states for subsystems LGSys, ENGSys and CPCS are requested. 2) Incoming state settings from actors <i>LGSys</i> , <i>ENGSys</i> and <i>CPCS</i> are stored. 3) Current indication states for FSB and RTS indication are requested from actors <i>FSB_CAB</i> and <i>FSB_PAX</i> . 4) Incoming indication states are stored. 5) Subsystem states and current indication states are evaluated with regard to FSB-RTS rules and customization settings (logical customization expressions, LCE).
Scenario Variation(s)	Scenario adapts to environment changes and can validate FSB-RTS indications at any time during aircraft operation.
Quality Constraint(s)	n/a
Postcondition(s)	FSB-RTS indication remains active / inactive as determined by subsystem states until all flight phases have been validated. In that case, FSB-RTS control for either <i>Cockpit_FSB_1</i> or <i>Cockpit_FSB_2</i> is set to off.
Why Essential?	Covered by most systems on the market, crucial service feature for cabin operations (stakeholder requirement), required for aircraft safety (legal requirement)

Passenger Notification System - Automated No Smoking Indication

Identifier	PNS-NS Signaling Auto
Actors	Subsystem actors <i>LGSys</i> , <i>ENGSys</i> and <i>CPCS</i> are part of the cabin system environment and are used for subsystem state representations. Human actors of the cockpit environment <i>Cockpit_NS_1</i> and <i>Cockpit_NS_2</i> represent cockpit crew members while human actor <i>NS_CAB</i> represents cabin crew members and <i>NS_PAX</i> represents passengers.
Purpose	During aircraft operation on ground or in flight, it is necessary to provide no smoking indications for passengers and cabin crew for safety reasons. An activated automated NS indication provides information depending on the status of different aircraft subsystems.

Included Service(s)	One service model is provided. This model describes the general use case intended for this AMIS model, i.e. an automated NS indication for cabin crew and passengers. In combination with different flight phases, this service is executed 16 times as part of the mission model described in chapter 5.4.1.
Trigger	One or more of the subsystem actors <i>LGSys</i> , <i>ENG Sys</i> or <i>CPCS</i> change their state while PNS-NS Signaling Auto is active.
Precondition(s)	Either actor <i>Cockpit_NS_1</i> or <i>Cockpit_NS_2</i> (customization dependent, logical actor assignment expression, LAAE) has activated automatic NS indication.
Expected Result	NS information is indicated correctly for each phase of aircraft operation.
Scenario	1) Subsystem states for subsystems <i>LGSys</i> , <i>ENG Sys</i> and <i>CPCS</i> are requested. 2) Incoming state settings from actors <i>LGSys</i> , <i>ENG Sys</i> and <i>CPCS</i> are stored. 3) Current indication states for NS indication are requested from actors <i>NS_CAB</i> and <i>NS_PAX</i> . 4) Incoming indication states are stored. 5) Subsystem states and current indication states are evaluated with regard to NS rules and customization settings (logical customization expressions, LCE).
Scenario Variation(s)	Scenario adapts to environment changes and can validate NS indications at any time during aircraft operation.
Quality Constraint(s)	n/a
Postcondition(s)	NS indication remains active / inactive as determined by subsystem states until all flight phases have been validated. In that case, NS control for either <i>Cockpit_NS_1</i> or <i>Cockpit_NS_2</i> is set to off.
Why Essential?	Covered by most systems on the market, crucial service feature for cabin operations (stakeholder requirement), required for aircraft safety (legal requirement)

B.2 MLDesigner Scenario Flowchart PAX Call

B.2.1 State Specification

Default Entrance in the FSM Top Level	
Description	
Action	
State 'IDLE' in the FSM Top Level	
Entry Action	
Exit Action	
State 'ACTIVE' in the FSM Top Level	
Entry Action	
Exit Action	
Default Entrance in State 'ACTIVE'	
Description	
Action	<pre> if (TimeOut > 0) {ScheduleEvent(Stop, (float)TimeOut, 0);} WriteMemory(Done, 0); WriteMemory(Runs, 0); WriteMemory(Flag, 0); WriteMemory(Counter, 0); WriteMemory(Counter2, 0); InsertFieldDS(Actor_DS, "GetCurrentState", "TRUE"); InsertFieldDS(Actor_DS, "NAME", (Kernel::String)PAX_Caller_Name); for (int i=1; i<=NumberOfPAXActors;i++) { InsertFieldDS(Actor_DS, "ID", i); WriteOutput(PAX_Caller_OUT, Actor_DS); } </pre>
State 'INITIALIZE_SCENARIO' in State 'ACTIVE'	
Entry Action	<pre> int idx = 0; if (DataNew(PAX_Caller_IN) == 1) { TypeRef actorDS = ReadNewInput(PAX_Caller_IN); int idx = (int)SelectFieldDS(actorDS, "ID"); TypeRef val = SelectFieldDS(actorDS, "Signal"); IncMemory(Counter); if (ReadIntMemory(Counter) < NumberOfPAXActors) { SetElementVector(PAX_Actors_Data, idx, val); } else if (ReadIntMemory(Counter) == NumberOfPAXActors) { SetElementVector(PAX_Actors_Data, idx, val); WriteMemory(Counter, 0); ScheduleEvent(Timer, 0, 0); } else { WriteMemory(Done, 0); WriteMemory(Validation_Information, "Error during initialization"); ScheduleEvent(Stop, (float)TimeOut, 0); } } </pre>
Exit Action	
State 'PRECONDITIONS' in State 'ACTIVE'	
Entry Action	<pre> if (TimeOut > 0) {CancelEvent(Stop, 0);ScheduleEvent(Stop, (float)TimeOut, 0);} if (DataNew(PAX_Caller_IN) == 1) { TypeRef actorDS = ReadNewInput(PAX_Caller_IN); int idx = (int)SelectFieldDS(actorDS, "ID"); TypeRef val = SelectFieldDS(actorDS, "Signal"); if ((EnumToString(SelectFieldDS(val, "PAX_Call_Indication")) == "inactive") && (AccessElementIntVector(Response_Actors, idx) != idx)) { SetElementIntVector(Response_Actors, idx, idx); IncMemory(Counter); } else </pre>

```

        {
            WriteMemory(Done, 0);

WriteMemory(Validation_Information, "Error during precondition PAX Caller Actor");
            ScheduleEvent(Stop, (float)TimeOut, 0);

        }
        if (ReadIntMemory(Counter) == NumberOfPAXActors)
        {
            WriteMemory(Counter, 0);
            IncMemory(Flag);
        }
    }
    if (DataNew(Cabin_Attendant_PAX_Call_Audible_IN) == 1) {
        TypeRef actorDS = ReadNewInput(Cabin_Attendant_PAX_Call_Audible_IN);
        int idx = (int)SelectFieldDS(actorDS, "ID");
        int idx_new = 60 + (int)SelectFieldDS(actorDS, "ID");
        TypeRef val = SelectFieldDS(actorDS, "Signal");

        if ((EnumToString(SelectFieldDS(val, "PAX_Call_Request")) == "inactive") && (AccessElementIntVector(Response_Actors, idx_new) != idx))
        {
            SetElementIntVector(Response_Actors, idx_new, idx);
            IncMemory(Counter2);
        }
        else
        {
            WriteMemory(Done, 0);
        }

WriteMemory(Validation_Information, "Error during precondition PAX Call CAB Audible Actor");
            ScheduleEvent(Stop, (float)TimeOut, 0);

        }
        if (ReadIntMemory(Counter2) == 3)
        {
            WriteMemory(Counter2, 0);
            IncMemory(Flag);
        }
    }
    if (DataNew(Cabin_Attendant_PAX_Call_Visual_IN) == 1) {
        TypeRef actorDS = ReadNewInput(Cabin_Attendant_PAX_Call_Visual_IN);
        int idx = (int)SelectFieldDS(actorDS, "ID");
        TypeRef val = SelectFieldDS(actorDS, "Signal");

        if ((EnumToString(SelectFieldDS(val, "PAX_Call_Request")) == "inactive") && (AccessElementIntVector(Response_Actors, 64) != idx))
        {
            SetElementIntVector(Response_Actors, 64, idx);
            IncMemory(Flag);
        }
        else
        {
            WriteMemory(Done, 0);
        }

WriteMemory(Validation_Information, "Error during precondition PAX Call CAB Visual Actor");
            ScheduleEvent(Stop, (float)TimeOut, 0);

        }
    }
    if (ReadIntMemory(Flag) >= 3)
    {
        WriteMemory(Flag, 0);
        ScheduleEvent(Timer, 0, 0);
    }
}

```

Exit Action

State 'A1_TRIGGER_PAX_CALL' in State 'ACTIVE'

Entry Action

```

if (TimeOut > 0) {CancelEvent(Stop, 0);ScheduleEvent(Stop, (float)TimeOut, 0);}
InsertFieldDS(Actor_DS, "GetCurrentState", "FALSE");
InsertFieldDS(Actor_DS, "NAME", (Kernel::String)PAX_Caller_Name);
InsertFieldDS(Actor_DS, "ID", ReadIntMemory(Scenario_Configuration));
InsertFieldDS(PAX_Caller_Actor_DS, "PAX_Call_Request", "Call");
InsertFieldDS(Actor_DS, "Signal", PAX_Caller_Actor_DS);
WriteOutput(PAX_Caller_OUT, Actor_DS);

```

Exit Action

State 'RI_PAX_CALL_INDICATIONS_ON' in State 'ACTIVE'

Entry Action

```

if (TimeOut > 0) {CancelEvent(Stop, 0);ScheduleEvent(Stop, (float)TimeOut, 0);}
if (DataNew(Cabin_Attendant_PAX_Call_Visual_IN) == 1) {
    TypeRef actorDS = ReadNewInput(Cabin_Attendant_PAX_Call_Visual_IN);
    TypeRef val = SelectFieldDS(actorDS, "Signal");
    TypeRef positionDS_in = SelectFieldDS(val, "PAX_Call_Position");

    TypeRef ActivePAXActorDS = AccessElementVector(PAX_Actors_Data, ReadIntMemory(Scenario_Configuration));

    TypeRef positionDS_current= SelectFieldDS(ActivePAXActorDS, "PAX_Call_Position");

    if ((EnumToString(SelectFieldDS(val, "PAX_Call_Request")) == "active") && SelectFieldDS(positionDS_in, "SeatRow") == SelectFieldDS(positionDS_current, "SeatRow") && SelectFieldDS(positionDS_in, "ZoneID") == SelectFieldDS(positionDS_current, "ZoneID") && SelectFieldDS(positionDS_in, "SeatID") == SelectFieldDS(positionDS_current, "SeatID"))
        {
            IncMemory(Flag);
        }
    else
    {
        WriteMemory(Done, 0);
    }

    WriteMemory(Validation_Information, "Error during PAX Caller Response Activate new PAX Call: CAB Visual");
    ScheduleEvent(Stop, (float)TimeOut, 0);
}
}
if (DataNew(PAX_Caller_IN) == 1) {
    TypeRef actorDS = ReadNewInput(PAX_Caller_IN);
    TypeRef val = SelectFieldDS(actorDS, "Signal");

    if ((EnumToString(SelectFieldDS(val, "PAX_Call_Indication")) == "active") && ((int)SelectFieldDS(actorDS, "ID") == ReadIntMemory(Scenario_Configuration)) && (EnumToString(SelectFieldDS(val, "PAX_CALL_INDICATION_Color")) == EnumToString(Color)))
        {
            IncMemory(Flag);
        }
    else
    {
        WriteMemory(Done, 0);
    }

    WriteMemory(Validation_Information, "Error during PAX Caller Response Activate new PAX Call: PAX Caller");
    ScheduleEvent(Stop, (float)TimeOut, 0);
}
}
if (DataNew(Cabin_Attendant_PAX_Call_Audible_IN) == 1) {
    TypeRef actorDS = ReadNewInput(Cabin_Attendant_PAX_Call_Audible_IN);
    int idx = (int)SelectFieldDS(actorDS, "ID");
    int idx_new = 60 + (int)SelectFieldDS(actorDS, "ID");
    TypeRef val = SelectFieldDS(actorDS, "Signal");

    if ((EnumToString(SelectFieldDS(val, "PAX_Call_Request")) == "active") && (AccessElementIntVector(Response_Actors, idx_new) != idx) && ((int)SelectFieldDS(val, "Audio_Characteristics") == 70))

```

```

        {
            SetElementIntVector(Response_Actors, idx_new, idx);
            IncMemory(Counter);
        }

else if ((EnumToString(SelectFieldDS(val, "PAX_Call_Request")) == "inactive") && (AccessElementIntVector(Response_Actors, idx_new) == idx))
    {
        IncMemory(Counter2);
    }
else
    {
        WriteMemory(Done, 0);

WriteMemory(Validation_Information, "Error during PAX Caller Response Activate new PAX Call: CAB Audible");
        ScheduleEvent(Stop, (float)TimeOut, 0);

    }
    if (ReadIntMemory(Counter) == 3 && ReadIntMemory(Counter2) == 3)
    {
        WriteMemory(Counter, 0);
        WriteMemory(Counter2, 0);
        IncMemory(Flag);
    }
}
if (ReadIntMemory(Flag) >= 3)
{
    WriteMemory(Flag, 0);
    ScheduleEvent(Timer, 0, 0);
}
}

```

Exit Action

State 'A2_WAIT_FOR_HELP' in State 'ACTIVE'

```

Entry Action int lb = SelectFieldDS(Wait_Objective, "Lower_Bound");
int ub = SelectFieldDS(Wait_Objective, "Upper_Bound");

double timetowait = rand() % lb + (ub-lb);

if (TimeOut > 0 && TimeOut < timetowait) {
    timetowait = rand() % (int)TimeOut;
}
ScheduleEvent(Timer, timetowait, 0);

```

Exit Action

State 'R2_NO_CHANGES' in State 'ACTIVE'

```

Entry Action if (TimeOut > 0) {CancelEvent(Stop, 0);ScheduleEvent(Stop, (float)TimeOut, 0);}
if (DataNew(Cabin_Attendant_PAX_Call_Audible_IN) == 1 || DataNew(Cabin_Attendant_PAX_Call_Visual_IN) == 1 || DataNew(PAX_Caller_IN) == 1 )
{
    WriteMemory(Done, 0);

WriteMemory(Validation_Information, "Error during PAX Caller Response Wait for no Changes");
        ScheduleEvent(Stop, (float)TimeOut, 0);
}

```

Exit Action

State 'A3_PAX_CALL_RESET' in State 'ACTIVE'

```

Entry Action if (TimeOut > 0) {CancelEvent(Stop, 0);ScheduleEvent(Stop, (float)TimeOut, 0);}
InsertFieldDS(PAX_Caller_Actor_DS, "PAX_Call_Request", "Reset");
InsertFieldDS(Actor_DS, "Signal", PAX_Caller_Actor_DS);
WriteOutput(PAX_Caller_OUT, Actor_DS);

```

Exit Action

State 'R3_PAX_CALL_INDICATIONS_OFF' in State 'ACTIVE'

```

Entry Action if (TimeOut > 0) {CancelEvent(Stop, 0);ScheduleEvent(Stop, (float)TimeOut, 0);}
if (DataNew(Cabin_Attendant_PAX_Call_Visual_IN) == 1) {
    TypeRef actorDS = ReadNewInput(Cabin_Attendant_PAX_Call_Visual_IN);
    TypeRef val = SelectFieldDS(actorDS, "Signal");
    TypeRef positionDS_in = SelectFieldDS(val, "PAX_Call_Position");

TypeRef ActivePAXActorDS = AccessElementVector(PAX_Actors_Data, ReadIntMemory(Scene_Configuration));
}

```

```

TypeRef positionDS_current= SelectFieldDS(ActivePAXActorDS, "PAX_Call_Position");

if ((EnumToString(SelectFieldDS(val, "PAX_Call_Request")) == "inactive") && SelectFieldDS(positionDS_in, "SeatRow") == SelectFieldDS(positionDS_current, "SeatRow") && SelectFieldDS(positionDS_in, "ZoneID") == SelectFieldDS(positionDS_current, "ZoneID") && SelectFieldDS(positionDS_in, "SeatID") == SelectFieldDS(positionDS_current, "SeatID"))
{
    IncMemory(Flag);
}
else
{
    WriteMemory(Done, 0);

WriteMemory(Validation_Information, "Error during Response End of PAX Call: CAB Visual");
    ScheduleEvent(Stop, (float)TimeOut, 0);
}
}
if (DataNew(PAX_Caller_IN) == 1) {
    TypeRef actorDS = ReadNewInput(PAX_Caller_IN);
    TypeRef val = SelectFieldDS(actorDS, "Signal");

if ((EnumToString(SelectFieldDS(val, "PAX_Call_Indication")) == "inactive") && ((int)SelectFieldDS(actorDS, "ID") == ReadIntMemory(Scenario_Configuration)))
{
    IncMemory(Flag);
}
else
{
    WriteMemory(Done, 0);

WriteMemory(Validation_Information, "Error during Response End of PAX Call: PAX Caller");
    ScheduleEvent(Stop, (float)TimeOut, 0);
}
}
if (DataNew(Cabin_Attendant_PAX_Call_Audible_IN) == 1) {
    WriteMemory(Done, 0);

WriteMemory(Validation_Information, "Error during Response End of PAX Call: CAB Audible received");
    ScheduleEvent(Stop, (float)TimeOut, 0);
}
if (ReadIntMemory(Flag) >= 2)
{
    WriteMemory(Flag, 0);
    ScheduleEvent(Timer, 0, 0);
}
}

```

Exit Action

State 'POST_CONDITIONS' in State 'ACTIVE'

```

Entry Action
if (TimeOut > 0) {CancelEvent(Stop, 0);ScheduleEvent(Stop, (float)TimeOut, 0);}
if (DataNew(PAX_Caller_IN) == 1) {
    TypeRef actorDS = ReadNewInput(PAX_Caller_IN);
    int idx = (int)SelectFieldDS(actorDS, "ID");
    TypeRef val = SelectFieldDS(actorDS, "Signal");

if ((EnumToString(SelectFieldDS(val, "PAX_Call_Indication")) == "inactive") && (AccessElementIntVector(Response_Actors, idx) != idx))
{
    SetElementIntVector(Response_Actors, idx, idx);
    IncMemory(Counter);
}
else
{
    WriteMemory(Done, 0);
}
}

```

```

WriteMemory(Validation_Information, "Error during postcondition PAX Caller Actor");
        ScheduleEvent(Stop, (float)TimeOut, 0);

    }
    if (ReadIntMemory(Counter) == NumberOfPAXActors)
    {
        WriteMemory(Counter, 0);
        IncMemory(Flag);
    }
}
if (DataNew(Cabin_Attendant_PAX_Call_Audible_IN) == 1) {
    TypeRef actorDS = ReadNewInput(Cabin_Attendant_PAX_Call_Audible_IN);
    int idx = (int)SelectFieldDS(actorDS, "ID");
    int idx_new = 60 + (int)SelectFieldDS(actorDS, "ID");
    TypeRef val = SelectFieldDS(actorDS, "Signal");

    if ((EnumToString(SelectFieldDS(val, "PAX_Call_Request")) == "inactive") && (AccessElementIntVector(Response_Actors, idx_new) != idx))
    {
        SetElementIntVector(Response_Actors, idx_new, idx);
        IncMemory(Counter2);
    }
    else
    {
        WriteMemory(Done, 0);
    }

    WriteMemory(Validation_Information, "Error during postcondition PAX Call CAB Audible Actor");
        ScheduleEvent(Stop, (float)TimeOut, 0);

    }
    if (ReadIntMemory(Counter2) == 3)
    {
        WriteMemory(Counter2, 0);
        IncMemory(Flag);
    }
}
if (DataNew(Cabin_Attendant_PAX_Call_Visual_IN) == 1) {
    TypeRef actorDS = ReadNewInput(Cabin_Attendant_PAX_Call_Visual_IN);
    int idx = (int)SelectFieldDS(actorDS, "ID");
    TypeRef val = SelectFieldDS(actorDS, "Signal");

    if ((EnumToString(SelectFieldDS(val, "PAX_Call_Request")) == "inactive") && (AccessElementIntVector(Response_Actors, 64) != idx))
    {
        SetElementIntVector(Response_Actors, 64, idx);
        IncMemory(Flag);
    }
    else
    {
        WriteMemory(Done, 0);
    }

    WriteMemory(Validation_Information, "Error during postcondition PAX Call CAB Visual Actor");
        ScheduleEvent(Stop, (float)TimeOut, 0);

    }
}
if (ReadIntMemory(Flag) >= 3)
{
    WriteMemory(Flag, 0);
    ScheduleEvent(Timer, 0, 0);
}
}

```

Exit Action

State 'PARAMETER_RECONFIGURATION' in State 'ACTIVE'

Entry Action

```

// validate each single PAX Caller
if (ReadIntMemory(Scenario_Configuration) < NumberOfPAXActors)
{

```

```

        IncMemory(Scenario_Configuration);
        WriteMemory(Done,0);
        ScheduleEvent(Timer,0, 0);
    }
    // end overall scenario
    else if ((ReadIntMemory(Scenario_Configuration) >= NumberOfPAXActors) && (AccessElementIntVector(Reconfiguration_Status, 1) == 3))
    {
        WriteMemory(Done,1);
        ScheduleEvent(Stop,0, 0);
    }
    //change customization
    else if ((AccessElementIntVector(Reconfiguration_Status, 2) == 1) && (AccessElementIntVector(Reconfiguration_Status, 1) < 3))
    {
        int enum_idx = AccessElementIntVector(Reconfiguration_Status, 1);
        int new_val = AccessElementIntVector(Reconfiguration_Status, 1) + 1;
        SetElementIntVector(Reconfiguration_Status, 1, new_val);
        IntToEnum(Color, enum_idx);
        InsertFieldDS(Customization_Configuration, "INDICATION_COLOR", Color);
        // Begin new PAX Caller run

        WriteOutput(PAX_Call_Customization_Actor_OUT, Customization_Configuration);

        WriteMemory(Scenario_Configuration, 1);
        ScheduleEvent(Timer,1, 0);
    }
}

```

Exit Action

State 'FINISHED' in the FSM Top Level

Entry Action WriteOutput(NumberOfRuns, Runs);

Exit Action

B.2.2 Transition Specification

```
Transition from State 'A1_TRIGGER_PAX_CALL' to State 'R1_PAX_CALL_INDICATIONS_ON'
Label
e
Event Expression
PAX_Caller_IN || Cabin_Attendant_PAX_Call_Visual_IN || Cabin_Attendant_PAX_Call_Audible_IN
Guard Condition
Action
Preemptive
No
Entry Type
History
Transition from State 'A2_WAIT_FOR_HELP' to State 'R2_NO_CHANGES'
Label
e
Event Expression
Guard Condition
Action
Preemptive
No
Entry Type
History
Transition from State 'A3_PAX_CALL_RESET' to State 'R3_PAX_CALL_INDICATIONS_OFF'
Label
e
Event Expression
PAX_Caller_IN || Cabin_Attendant_PAX_Call_Visual_IN || Cabin_Attendant_PAX_Call_Audible_IN
Guard Condition
Action
Preemptive
No
Entry Type
History
Transition from State 'ACTIVE' to State 'FINISHED'
Label
Stop / OK
Event Expression
Stop
Guard Condition
ReadIntMemory(Done) == 1
Action
StringList Result;
Result = "Scenario Completed with no errors (Number of validated PAX Call Customizations: ";
Result << (Kernel::String)ReadIntMemory(Runs);
Result << ")";
WriteMemory(Validation_Information, Result);
WriteOutput(End, 1);
Preemptive
No
Entry Type
History
Transition from State 'ACTIVE' to State 'FINISHED'
Label
Stop / FAILED
Event Expression
Stop
Guard Condition
ReadIntMemory(Done) == 0
Action
WriteMemory(Validation_Information, "Error During Validation");
WriteOutput(End, 0);
Preemptive
No
Entry Type
History
Transition from State 'FINISHED' to State 'IDLE'
Label
e / D
Event Expression
Guard Condition
Action
Preemptive
No
Entry Type
History
Transition from State 'IDLE' to State 'ACTIVE'
Label
t
Event Expression
t
Guard Condition
Action
Preemptive
No
Entry Type
History
Transition from State 'INITIALIZE_SCENARIO' to State 'INITIALIZE_SCENARIO'
Label
PAX_Caller_IN
Event Expression
PAX_Caller_IN
Guard Condition
Action
Preemptive
No
Entry Type
History
Transition from State 'INITIALIZE_SCENARIO' to State 'PRECONDITIONS'
Label
A
Event Expression
Timer
Guard Condition
```

```

Action //CAB Visual
InsertFieldDS(Actor_DS, "GetCurrentState", "TRUE");
InsertFieldDS(Actor_DS, "NAME", (Kernel::String)CAB_Visual_Name);
InsertFieldDS(Actor_DS, "ID", 1);
WriteOutput(Cabin_Attendant_PAX_Call_Visual_OUT, Actor_DS);
//CAB Audible
for (int i=1; i<=3;i++)
{
    InsertFieldDS(Actor_DS, "NAME", (Kernel::String)CAB_Audible_Name);
    InsertFieldDS(Actor_DS, "ID", i);
    WriteOutput(Cabin_Attendant_PAX_Call_Audible_OUT, Actor_DS);
}
//PAX Caller
for (int i=1; i<=NumberOfPAXActors;i++)
{
    InsertFieldDS(Actor_DS, "NAME", (Kernel::String)PAX_Caller_Name);
    InsertFieldDS(Actor_DS, "ID", i);
    WriteOutput(PAX_Caller_OUT, Actor_DS);
}
IncMemory(Runs);
Preemptive No
Entry Type History
Transition from State 'PARAMETER_RECONFIGURATION' to State 'PRECONDITIONS'
Label e
Event Expression Timer
Guard Condition ReadIntMemory(Done) != 1
Action //CAB Visual
InsertFieldDS(Actor_DS, "GetCurrentState", "TRUE");
InsertFieldDS(Actor_DS, "NAME", (Kernel::String)CAB_Visual_Name);
InsertFieldDS(Actor_DS, "ID", 1);
WriteOutput(Cabin_Attendant_PAX_Call_Visual_OUT, Actor_DS);
//CAB Audible
for (int i=1; i<=3;i++)
{
    InsertFieldDS(Actor_DS, "NAME", (Kernel::String)CAB_Audible_Name);
    InsertFieldDS(Actor_DS, "ID", i);
    WriteOutput(Cabin_Attendant_PAX_Call_Audible_OUT, Actor_DS);
}
//PAX Caller
for (int i=1; i<=NumberOfPAXActors;i++)
{
    InsertFieldDS(Actor_DS, "NAME", (Kernel::String)PAX_Caller_Name);
    InsertFieldDS(Actor_DS, "ID", i);
    WriteOutput(PAX_Caller_OUT, Actor_DS);
}
IncMemory(Runs);
Preemptive No
Entry Type History
Transition from State 'POST_CONDITIONS' to State 'POST_CONDITIONS'
Label Cabin_Attendant_PAX_Call_Visual_IN ||
Cabin_Attendant_PAX_Call_Audible_IN ||
PAX_Caller_IN
Event Expression PAX_Caller_IN || Cabin_Attendant_PAX_Call_Visual_IN || Cabin_Attendant_PAX_Call_Audible_IN
Guard Condition
Action
Preemptive No
Entry Type History
Transition from State 'POST_CONDITIONS' to State 'PARAMETER_RECONFIGURATION'
Label F
Event Expression Timer
Guard Condition
Action for (int i=0; i<65;i++)
{
    SetElementIntVector(Response_Actors, i, 0);
}
InsertFieldDS(Actor_DS, "GetCurrentState", "FALSE");
if (TimeOut > 0) {CancelEvent(Stop, 0);}
Preemptive No
Entry Type History
Transition from State 'PRECONDITIONS' to State 'PRECONDITIONS'
Label Cabin_Attendant_PAX_Call_Visual_IN ||
Cabin_Attendant_PAX_Call_Audible_IN ||

```

```

PAX Caller_IN
Event Expression PAX Caller_IN || Cabin_Attendant_PAX_Call_Visual_IN || Cabin_Attendant_PAX_Call_Audible_IN
Guard Condition
Action
Preemptive No
Entry Type History
Transition from State 'PRECONDITIONS' to State 'A1_TRIGGER_PAX_CALL'
Label B
Event Expression Timer
Guard Condition
Action for (int i=0; i<65;i++)
{
    SetElementIntVector(Response_Actors, i, 0);
}
Preemptive No
Entry Type History
Transition from State 'R1_PAX_CALL_INDICATIONS_ON' to State 'R1_PAX_CALL_INDICATIONS_ON'
Label Cabin_Attendant_PAX_Call_Visual_IN || Cabin_Attendant_PAX_Call_Audible_IN || PAX Caller_IN
Event Expression PAX Caller_IN || Cabin_Attendant_PAX_Call_Visual_IN || Cabin_Attendant_PAX_Call_Audible_IN
Guard Condition
Action
Preemptive No
Entry Type History
Transition from State 'R1_PAX_CALL_INDICATIONS_ON' to State 'A2_WAIT_FOR_HELP'
Label C
Event Expression Timer
Guard Condition
Action for (int i=0; i<65;i++)
{
    SetElementIntVector(Response_Actors, i, 0);
}
Preemptive No
Entry Type History
Transition from State 'R2_NO_CHANGES' to State 'R2_NO_CHANGES'
Label Cabin_Attendant_PAX_Call_Visual_IN || Cabin_Attendant_PAX_Call_Audible_IN || PAX Caller_IN
Event Expression PAX Caller_IN || Cabin_Attendant_PAX_Call_Visual_IN || Cabin_Attendant_PAX_Call_Audible_IN
Guard Condition
Action
Preemptive No
Entry Type History
Transition from State 'R2_NO_CHANGES' to State 'A3_PAX_CALL_RESET'
Label D
Event Expression Timer
Guard Condition
Action
Preemptive No
Entry Type History
Transition from State 'R3_PAX_CALL_INDICATIONS_OFF' to State 'R3_PAX_CALL_INDICATIONS_OFF'
Label Cabin_Attendant_PAX_Call_Visual_IN || Cabin_Attendant_PAX_Call_Audible_IN || PAX Caller_IN
Event Expression PAX Caller_IN || Cabin_Attendant_PAX_Call_Visual_IN || Cabin_Attendant_PAX_Call_Audible_IN
Guard Condition
Action
Preemptive No
Entry Type History
Transition from State 'R3_PAX_CALL_INDICATIONS_OFF' to State 'POST_CONDITIONS'
Label E
Event Expression Timer
Guard Condition
Action //CAB Visual
InsertFieldDS(Actor_DS, "GetCurrentState", "TRUE");
InsertFieldDS(Actor_DS, "NAME", (Kernel::String)CAB_Visual_Name);
InsertFieldDS(Actor_DS, "ID", 1);
WriteOutput(Cabin_Attendant_PAX_Call_Visual_OUT, Actor_DS);

```

```
//CAB Audible
for (int i=1; i<=3;i++)
{
    InsertFieldDS(Actor_DS, "NAME", (Kernel::String)CAB_Audible_Name);
    InsertFieldDS(Actor_DS, "ID", i);
    WriteOutput(Cabin_Attendant_PAX_Call_Audible_OUT, Actor_DS);
}
//PAX Caller
for (int i=1; i<=NumberOfPAXActors;i++)
{
    InsertFieldDS(Actor_DS, "NAME", (Kernel::String)PAX Caller_Name);
    InsertFieldDS(Actor_DS, "ID", i);
    WriteOutput(PAX Caller_OUT, Actor_DS);
}
```

Preemptive

No

Entry Type

History

B.3 Overall Data and Customization Model

Root.SIMKAB_DS.BlinkCharacteristics			
BLINK_DURATION	Root.ENUM.Duration		s2
BLINK_FREQUENCY	Root.ENUM.Frequency		hz1
Root.SIMKAB_DS.Cabin_Chime			
Source_ID	Root.Integer	[0, Inf]	0
Chime_Function_Source	Root.ENUM.CabinChime_SourceType		PA_Chime
Chime_Type	Root.ENUM.CabinChime_SubType		PA_Chime_1
Destination_Zones	Root.IntVector		10:0
Root.SIMKAB_DS.Cabin_Chime_Audio			
Source_ID	Root.Integer		0
Type	Root.ENUM.CabinChime_SourceType		PA_Chime
Volume	Root.Integer	[20, 100]	60
Duration	Root.Float	[5, 30]	30
Root.SIMKAB_DS.CAB_PAX_Call_Audible_Actor			
PAX_Call_Request	Root.ENUM.Call_Indication		inactive
Audio_Characteristics	Root.Integer		0
Root.SIMKAB_DS.CAB_PAX_Call_Visual_Actor			
PAX_Call_Request	Root.ENUM.Call_Indication		inactive
PAX_Call_Position	Root.SIMKAB_DS.Call_Position		
Root.SIMKAB_DS.CAB_PA_Actor			
PA_Data	Root.SIMKAB_DS.PA		
PA_HMI_Data	Root.SIMKAB_DS.PA_HMI		
Root.SIMKAB_DS.CAB_RDY			
Cabin_Ready_for_TakeOff_or_Landing	Root.ENUM.Boolean		FALSE
CPT_Acknowledge	Root.ENUM.Boolean		FALSE
System_Sender_Flag	Root.Integer	[0, 1]	0
Root.SIMKAB_DS.CAB_to_CAB_InterCom_Actor			
InterComData	Root.SIMKAB_DS.Intercom_Request		
InterComAudio	Root.SIMKAB_DS.InterCom_AUDIO_Stream		
InterComHMI	Root.SIMKAB_DS.InterCom_HMI		
InterComChime	Root.SIMKAB_DS.Cabin_Chime		
Data_ID	Root.Integer		0
Root.SIMKAB_DS.Call_Position			
SeatID	Root.Integer		0
SeatRow	Root.Integer	[1, 1000]	1
ZoneID	Root.Integer	[0, 9]	0
Root.SIMKAB_DS.Environment_Info			
Aircraft_On_Ground	Root.ENUM.Boolean		FALSE
Root.SIMKAB_DS.InterCom_AUDIO_Stream			
InterCom_Audio_Source_ID	Root.Integer	[0, Inf]	0
InterCom_Audio_Destination_IDs	Root.IntVector		40:-111
InterCom_Audio_Destination	Root.Integer	[-1, Inf]	0
Start_Time	Root.Float	[0, Inf]	0
Duration	Root.Float	[0, Inf]	0
Audio_Frequency	Root.Float	[8000, 32000]	16000
Quantization	Root.Integer	[8, 32]	16
Max_Latency_Source2Sink	Root.Float	[0, 32]	0
Max_Jitter	Root.Float	[0, 4]	0
Volume_Level_Db	Root.Float	[0, 60]	20
Muted	Root.ENUM.YesNo		No
End_Call	Root.ENUM.YesNo		No
Root.SIMKAB_DS.InterCom_Env			
Aircraft_On_Ground	Root.ENUM.Boolean		TRUE
Root.SIMKAB_DS.InterCom_HMI			
Incomming Caller_ID	Root.Integer	[0, Inf]	0
Outgoing_Destination_ID	Root.Integer	[-1, Inf]	0
Light_Indication	Root.ENUM.InterCom_Indication_Modes		CAB_Flashing_Green_Light
EMERGENCY_Call	Root.Integer	[0, 1]	0
CPT_Call	Root.Integer	[0, 1]	0
SERVICE_Call	Root.Integer	[0, 1]	0
CAB_Call	Root.Integer	[0, 1]	0
Call_Priority	Root.Integer	[1, 3]	3

Call_Established	Root.Integer	[0, 1]	0	
Addressee_Engaged	Root.Integer	[0, 1]	0	
Call_Declined	Root.Integer	[0, 1]	0	
Call_Active	Root.Integer	[0, 1]	0	
Call_Ended	Root.Integer	[0, 1]	0	
Reset_HMI	Root.Integer	[0, 1]	0	
Root.SIMKAB_DS.Intercom_Request				
InterCom_Source	Root.Integer	[0, Inf]	0	
Call_Type	Root.ENUM.InterCom_Types			Cabin to Cabin Call
Call_Priority	Root.Integer	[1, 3]	3	
Call_Destination	Root.Integer	[-1, Inf]	0	
Call_Request	Root.ENUM.InterCom_Request			Initiate New Call
AudioPermission	Root.ENUM.YesNo			No
My_Aircraft_Zone	Root.IntVector			10:0
Root.SIMKAB_DS.LAV_Smoke_Audible				
Chime_Status	Root.ENUM.LAV_Chime			chime_inactive
Chime_Characteristics	Root.SIMKAB_DS.LAV_SMOKE_CHIME_CHARACTERISTICS			
Root.SIMKAB_DS.LAV_SMOKE_CHIME_CHARACTERISTICS				
CHIME_TYPE	Root.ENUM.LAV_Chime_Type			Type1
VOLUME_dB	Root.Integer	[30, 80]	50	
Root.SIMKAB_DS.LAV_Smoke_Customization_DS				
CABIN_CHIME_ACTIVATION_WHEN_LAV_SMOKE	Root.ENUM.Boolean			FALSE
LAV_SMOKE_CHIME_CHARACTERISTICS	Root.SIMKAB_DS.LAV_SMOKE_CHIME_CHARACTERISTICS			
Root.SIMKAB_DS.LAV_Smoke_ENV				
Smoke_State	Root.ENUM.Smoke_State			smoke
Indication_Location	Root.SIMKAB_DS.LAV_SMOKE_LOCATION			
Root.SIMKAB_DS.LAV_SMOKE_LOCATION				
Area	Root.ENUM.Classes			Economy
LAV	Root.ENUM.LAVs			LAV1
Root.SIMKAB_DS.LAV_Smoke_Visual				
Indication_State	Root.ENUM.Active_State			inactive
Indication_Location	Root.SIMKAB_DS.LAV_SMOKE_LOCATION			
Root.SIMKAB_DS.Light_CMS_RL				
Reading_Lights_Active	Root.ENUM.Boolean			FALSE
Root.SIMKAB_DS.PA				
PA_Source_ID	Root.Integer			1
PA_Request	Root.ENUM.PA_Request			Initiate_PA
PA_Priority	Root.ENUM.PA_Priority			Normal_Cabin_PA
AudioPermission	Root.ENUM.YesNo			No
Zone_1_Direct_PA	Root.Integer	[0, 1]	0	
Zone_2_PA_All	Root.Integer	[0, 1]	0	
Zone_3_FirstClass	Root.Integer	[0, 1]	0	
Zone_4_BusinessClass	Root.Integer	[0, 1]	0	
Zone_5_EconomyClass	Root.Integer	[0, 1]	0	
Zone_6_Lavatories	Root.Integer	[0, 1]	0	
Zone_7_Doors	Root.Integer	[0, 1]	0	
Zone_8_Galleys	Root.Integer	[0, 1]	0	
Zone_9_CrewRestCompartments	Root.Integer	[0, 1]	0	
PA_Override_Info	Root			
Root.SIMKAB_DS.PAX_Call				
PAX_ID	Root.Integer	(-1, Inf)	1	
Name	Root.String			abc
Priority	Root.Integer	(0, 4)	3	
Root.SIMKAB_DS.PAX_Caller_Actor				
PAX_Call_Request	Root.ENUM.Call_Request			Call
PAX_Call_Position	Root.SIMKAB_DS.Call_Position			
PAX_Call_Indication	Root.ENUM.Call_Indication			inactive
PAX_CALL_INDICATION_Color	Root.ENUM.PAX_CALL_INDICATION_COLOR			blue
PAX_CAL_INDICATION_Mode	Root.SIMKAB_DS.PAX_CALL_INDICATION_MODE			
Root.SIMKAB_DS.PAX_Call_Customization_DS				
INDICATION_COLOR	Root.ENUM.PAX_CALL_INDICATION_COLOR			blue
INDICATION_MODE	Root.SIMKAB_DS.PAX_CALL_INDICATION_MODE			
Root.SIMKAB_DS.PAX_CALL_INDICATION_MODE				
MODE	Root.ENUM.IndicationMode			steady
PAX_LIGHT_CHARACTERISTICS	Root.SIMKAB_DS.BlinkCharacteristics			
Root.SIMKAB_DS.PAX_Call_Inhibit				
Inhibit_All	Root.ENUM.Boolean			FALSE

Allow_All	Root.ENUM.Boolean		FALSE
Inhibit_Zone	Root.Integer	[-1, 9]	-1
Allow_Zone	Root.Integer	[-1, 9]	-1
Inhibit_Seat_Row	Root.Integer	[-1, 1000]	-1
Allow_Seat_Row	Root.Integer	[-1, 1000]	-1
Inhibit_Distinct_ID	Root.SIMKAB_DS.PAX_Call		
Allow_Distinct_ID	Root.SIMKAB_DS.PAX_Call		
Root.SIMKAB_DS.PAX_Call_Request			
PAX_ID	Root.Integer	(-1, Inf)	0
Zone_ID	Root.Integer	[0, 9]	0
Seat_Row	Root.Integer	[1, 1000]	1
PAX_Call_DS	Root.SIMKAB_DS.PAX_Call		
Name	Root.String		abc
PAX_Call_Priority	Root.Integer	(0, 4)	3
Active	Root.ENUM.Boolean		FALSE
Root.SIMKAB_DS.PAX_Call_Reset			
PAX_ID	Root.Integer	(-1, Inf)	0
Zone_ID	Root.Integer	[0, 9]	0
Seat_Row	Root.Integer	[0, 1000]	0
Clear_PAX_DS	Root.SIMKAB_DS.PAX_Call		
Clear_All	Root.Integer		0
Root.SIMKAB_DS.PAX_Call_Reset_Confirmation			
Call_Clearance_DS	Root.SIMKAB_DS.PAX_Call		
Reset_Clearance	Root.SIMKAB_DS.PAX_Call		
Root.SIMKAB_DS.PAX_HMI			
PAX_ID	Root.Integer	(-1, Inf)	0
Zone_ID	Root.Integer	[0, 9]	0
Seat_Row	Root.Integer	[1, 1000]	1
Name	Root.String		abc
PAX_Call_Priority	Root.Integer	(0, 4)	3
Active	Root.ENUM.Boolean		FALSE
Root.SIMKAB_DS.PAX_PA_Actor			
PAX_PA_Audio	Root.SIMKAB_DS.PA_AUDIO_Stream		
Root.SIMKAB_DS.PA_AUDIO_Stream			
PA_Source_ID	Root.Integer		0
x_y_z_LOCATION_PA_Source	Root.FloatVector		3:0.00000
Destination_Zones	Root.IntVector		10:0
StartTime	Root.Float	[0, Inf)	0
Duration	Root.Float	[0, Inf)	0
Audio_Frequency	Root.Float	[22000, 44000]	32000
Quantization	Root.Integer	[8, 16]	16
Max_Latency_Source2Sink	Root.Float	[0, 32]	32
Max_Jitter	Root.Float	[0, 4]	4
Volume_Level_Db	Root.Float	[60, 100]	60
Muted	Root.ENUM.YesNo		No
Abort_or_End_PA	Root.ENUM.YesNo		No
Root.SIMKAB_DS.PA_Control_Response			
Data	Root.SIMKAB_DS.PA		
Root.SIMKAB_DS.PA_HMI			
PA_Info	Root.ENUM.PA_HMI_Info		New_PA_Granted
PA_Source_ID	Root.Integer		0
PA_Priority	Root.ENUM.PA_Priority		Cockpit_PA
PA_Zones	Root.SIMKAB_DS.PA		
Root.SIMKAB_DS.PA_HMI_CPT			
Data	Root.SIMKAB_DS.PA_HMI		
Root.SIMKAB_DS.PA_Status			
Get_Status	Root.ENUM.YesNo		No
PAs_Active	Root.ENUM.YesNo		No
PA_Zones_Status	Root.Vector		
Root.SIMKAB_DS.PLS_Control_Actor			
FSB_AUTO_Control	Root.ENUM.PLS_Control		off
Root.SIMKAB_DS.PLS_CPCS_Actor			
Status	Root.ENUM.Boolean		FALSE
Root.SIMKAB_DS.PLS_ENGSys_Actor			
Status	Root.ENUM.Boolean		FALSE
Root.SIMKAB_DS.PLS_FSB_AUTO_Customization			

AUTO Conditions	Root.ENUM.FSB_AUTO_CFG		Landing Gear Down and Locked
FSB_Control_Assignment	Root.ENUM.FSB_Control		Cockpit_FSB_1
Root.SIMKAB_DS.PLS_Indication_Actor			
Indication_Status	Root.ENUM.FSB_Indication		inactive
Root.SIMKAB_DS.PLS_LGSys_Actor			
Status	Root.ENUM.Boolean		FALSE
Root.SIMKAB_DS.PLS_NS_AUTO_Customization			
AUTO Conditions	Root.ENUM.FSB_AUTO_CFG		Landing Gear Down and Locked
NS_Control_Assignment	Root.ENUM.NS_Control		Cockpit_NS_1
Root.SIMKAB_DS.PNS_Control_FSB_RTS			
FSB_RTS_Setting	Root.ENUM.PNS_Switch_Mode		OFF
Root.SIMKAB_DS.PNS_Control_NS			
NS_Setting	Root.ENUM.PNS_Switch_Mode		OFF
Root.SIMKAB_DS.PNS_Env			
Aircraft_On_Ground	Root.ENUM.Boolean		TRUE
Normal_Flight_Phase	Root.ENUM.Boolean		FALSE
Landing_Gear_Down_and_Locked	Root.ENUM.Boolean		FALSE
Aircraft_Height_Descent_Mode	Root.ENUM.Boolean		FALSE
Loss_Of_Cabin_Pressure	Root.ENUM.Boolean		FALSE
Root.SIMKAB_DS.PNS_FS_RTS_Management			
FSB_Sign_Active	Root.ENUM.Boolean		FALSE
RTS_Sign_Active	Root.ENUM.Boolean		FALSE
FSB_Sign_Configuration	Root.SIMKAB_DS.Sub_PNS		
RTS_Sign_Configuration	Root.SIMKAB_DS.Sub_PNS		
Root.SIMKAB_DS.PNS_HMI			
PNS_Sign_Type	Root.ENUM.PNS_Sign_Type		FSB
Sign_Active	Root.ENUM.Boolean		FALSE
Permanent_ON	Root.ENUM.YesNo		No
Permanent_OFF	Root.ENUM.YesNo		No
Sign_Configuration	Root.SIMKAB_DS.Sub_PNS		
Root.SIMKAB_DS.PNS_NS_Management			
NS_Sign_Active	Root.ENUM.Boolean		FALSE
NS_Sign_Configuration	Root.SIMKAB_DS.Sub_PNS		
Root.SIMKAB_DS.Probe			
ID	Root.Integer	[0, Inf]	0
Colum_Index	Root.Integer	[0, Inf]	0
Data_as_String	Root.String		""
Name	Root.String		""
Data	Root		
Root.SIMKAB_DS.SaD_Env			
All_Cabin_Doors_Locked	Root.ENUM.Boolean		FALSE
All_Slides_Armed	Root.ENUM.Boolean		FALSE
Door_IDS_Not_Locked	Root.IntVector		20:0
Slide_IDS_Not_Armed	Root.IntVector		20:0
Root.SIMKAB_DS.SaD_HMI			
Slides_Armed_Doors_Locked	Root.ENUM.Boolean		FALSE
Door_ID_Unlocked	Root.Integer		0
Slide_ID_Unarmed	Root.Integer		0
Root.SIMKAB_DS.Service_InterCom_Actor			
InterComData	Root.SIMKAB_DS.Intercom_Request		
InterComAudio	Root.SIMKAB_DS.InterCom_AUDIO_Stream		
InterComHMI	Root.SIMKAB_DS.InterCom_HMI		
Data_ID	Root.Integer		0
Root.SIMKAB_DS.Sub_PNS			
Sign_Mode	Root.ENUM.PNS_Sign_Modes		Continuous_Light
Flash_Duration	Root.Integer	[0, 20]	5
Flash_Frequency	Root.Float	[0.5, 2]	1
Pulsing_Frequency	Root.Float	[0.5, 2]	0.5
Light_Color_R	Root.Integer	[0, 255]	0
Light_Color_G	Root.Integer	[0, 255]	0
Light_Color_B	Root.Integer	[0, 255]	0
Root.ENUM.CabinChime_SourceType			
PA_Chime			0
PAX_Call			1
InterCom			2
PNS_Chime			3

Root.ENUM.CabinChime_SubType	
PA_Chime_1	0
PA_Chime_2	1
PAX_Call_Chime_1	2
PAX_Call_Chime_2	3
InterCom_EMERGENCY_Chime	4
InterCom_COCKPIT_Chime	5
InterCom_SERVICE_Chime	6
InterCom_CABIN_Chime	7
PNS_Chime_1	8
PNS_Chime_2	9

Root.ENUM.Call_Indication	
inactive	0
active	1

Root.ENUM.Call_Request	
Call	0
Reset	1

Root.ENUM.Decompression_Status	
NO_Decompression	0
PARTIAL_Decompression	1
FULL_Decompression	2
Decompression_Status_FAILURE	3

Root.ENUM.Duration	
s2	0
s5	1
s10	2

Root.ENUM.Emergency_Status	
Emergency_INACTIVE	0
Emergency_ACTIVE	1
Emergency_Status_FAILURE	2

Root.ENUM.Engine_Operation_Status	
ALL_Engines_OFF	0
ALL_Engines_ON_Level1	1
ALL_Engines_ON_Level2	2
ALL_Engines_ON_Level3	3
SINGLE_Engine_Operation	4
Engine_Status_FAILURE	5

Root.ENUM.Flight_Phase_Status	
Preflight	0
Taxi_Out	1
Take_Off_to80kts	2
Take_Off_V1	3
Take_Off_to_LiftOff	4
Climb_to400ft	5
Climb_to1500ft	6
Cruise	7
Emergency_Descent	8
Approach	9
Landing	10
Go_Around	11
Taxi_In_Start	12
Taxi_In_Leave_Runway	13
Post_Flight_AllEnginesOff	14
Flight_Phase_FAILURE	15

Root.ENUM.Frequency	
hz1	0
hz2	1
hz5	2

Root.ENUM.FSB_AUTO_CFG	
Landing Gear Down and Locked	0
Landing Gear Down and Locked AND Engines ON AND notCRUISE_Mode	1
Landing Gear Down and Locked OR Loss of Cabin Pressure	2
Root.ENUM.FSB_Control	
Cockpit FSB 1	0
Cockpit FSB 2	1
Cockpit FSB 1 OR Cockpit FSB 2	2
Root.ENUM.FSB_Indication	
inactive	0
active	1
Root.ENUM.IndicationMode	
steady	0
blinking	1
Root.ENUM.InterCom_Indication_Modes	
EMER Flashing Red Light	0
CPT Flashing Blue Light	1
SERVICE Flashing Yellow Light	2
CAB Flashing Green Light	3
Root.ENUM.InterCom_Request	
Initiate New Call	0
Abort or End Call	1
Call Declined	2
Addressee Engaged	3
Call Accepted	4
Connection Established	5
Get Station ID	6
Call Timed Out	7
Root.ENUM.Intercom_Station_Type	
Cockpit_Station	0
Ground_Service_Station	1
Cabin_Station	2
Root.ENUM.InterCom_Types	
Emergency_Call	0
Cockpit_to_Cabin_Call	1
Ground_Service_to_Cabin_Call	2
Cabin_to_Ground_Service_Call	3
Cabin_to_Cockpit_Call	4
Cabin_to_Cabin_Call	5
Root.ENUM.NS_Control	
Cockpit_NS_1	0
Cockpit_NS_2	1
Cockpit_NS_1_OR_Cockpit_NS_2	2
Root.ENUM.PAX_CALL_INDICATION_COLOR	
blue	0
green	1
yellow	2
Root.ENUM.PA_HMI_Info	
New_PA_Granted	0
Active_PA_Aborted_or_Ended	1
Active_PA_Override_Suspended	2
Waiting_PA_Granted	3
New_HigherPriority_PA_Granted	4
Root.ENUM.PA_Priority	
Cockpit_PA	0

Priority_Cabin_PA	1
Normal_Cabin_PA	2
Entertainment_PA	3

Root.ENUM.PA_Request	
Initiate_PA	0
Abort_or_End_PA	1
PA_Timed_Out	2
PA_Override	3
PA_Priority_Wait	4
PA_Request_Invalid	5
PA_Audio_Quality_Invalid	6

Root.ENUM.PLS_Control	
off	0
on	1

Root.ENUM.PNS_Sign_Modes	
Continuous_Light	0
Flashing_Light	1
Pulsing_Light	2

Root.ENUM.PNS_Sign_Type	
FSB	0
RTS	1
NS	2

Root.ENUM.PNS_Switch_Mode	
ON	0
AUTO	1
OFF	2

Root.ENUM.Seat	
A	0
B	1
C	2

Root.ENUM.Smoke_State	
smoke	0
no_smoke	1

Root.ENUM.Classes	
First	0
Business	1
Economy	2

Root.ENUM.LAVs	
LAV1	0
LAV2	1
LAV3	2
LAV4	3
LAV5	4
LAV6	5
LAV7	6
LAV8	7
LAV9	8

Root.ENUM.LAV_Chime	
chime_in_progress	0
chime_inactive	1

Root.ENUM.LAV_Status	
OK	0
NOT_OK	1
TEST	2
FAILURE	3

B.4 Generated Validation Report

Automatically Generated System Validation Report

Title: Validation Report Cabin Management System
 Validation Run: 4.0
 Author(s): Nils Fischer
 Date and time of generation: Fri Aug 05 17:46:24 2016

Overall Validation Status: Validated

System: CMS
 System ID: 1
 Mission: CMSMISSION
 Mission ID: 1
 Atomic Mission: n/a
 Atomic Mission ID: n/a
 Quality Objective Mission: Weight
 Service: n/a
 Service ID: n/a
 Scenario: n/a
 Scenario ID: n/a
 Requirement Type: Non_Functional
 Information: Non-functional parameter value is higher than the lower boundary and less or equal than the mean of the objective budget.
 (Overall mission budget is: Root.ObjectiveWeight,0,400,600,0.4);

Weight:
 Mean Value: 278
 Lower Boundary: 214
 Upper Boundary: 350

Validation Status: Validated

System: CMS
 System ID: 1
 Mission: CMSMISSION
 Mission ID: 1
 Atomic Mission: n/a
 Atomic Mission ID: n/a
 Quality Objective Mission: Cost
 Service: n/a
 Service ID: n/a
 Scenario: n/a
 Scenario ID: n/a
 Requirement Type: Non_Functional
 Information: Non-functional parameter value is less or equal than the upper boundary of the objective budget.
 (Overall mission budget is: Root.ObjectiveCost,0,20000,80000,0.4);

Cost:
 Mean Value: 43100
 Lower Boundary: 25400
 Upper Boundary: 62000

Validation Status: Validated

System: CMS
 System ID: 1
 Mission: CMSMISSION
 Mission ID: 1
 Atomic Mission: Cabin_Ground_ServiceInterCom
 Atomic Mission ID: 1
 Quality Objective Mission: nA
 Service: Cabin_Ground_ServiceInterCom_Service
 Service ID: 1
 Scenario: ServiceInterCom_Scenario_ModelA
 Scenario ID: 1
 Requirement Type: Mixed
 Information: Scenario Completed with no errors; NF Objectives: (Audio_Frequency, 8000, 8000, 32000), (Audio_Quantization, 8, 8, 32), (Volume_Level, 0, 50, 60), (Audio_Latency, 0, 6, 32);

Validation Status: Validated

System: CMS
 System ID: 1
 Mission: CMSMISSION
 Mission ID: 1
 Atomic Mission: Cabin_Cabin_InterCom
 Atomic Mission ID: 2
 Quality Objective Mission: nA
 Service: Cabin_InterCom_Service
 Service ID: 1
 Scenario: InterCom_Scenario_ModelA

Scenario ID: 1
Requirement Type: Mixed
Information: Scenario Completed with no errors; NF Objectives: (Audio.Frequency, 8000, 16000, 32000), (Audio.Quantization, 8,16, 32), (Volume.Level, 0, 40, 60), (Audio.Latency, 0, 4, 32);

Validation Status: Validated

System: CMS
System ID: 1
Mission: CMSMISSION
Mission ID: 1
Atomic Mission: AMIS_Public_Address
Atomic Mission ID: 3
Quality Objective Mission: nA
Service: PA_Service_Cabin
Service ID: 1
Scenario: PA_Scenario_ModelA
Scenario ID: 1
Requirement Type: Mixed
Information: Scenario Completed with no errors (Number of different PA combinations validated: 1677);
Validation Status: Validated

System: CMS
System ID: 1
Mission: CMSMISSION
Mission ID: 1
Atomic Mission: PAX_Call
Atomic Mission ID: 4
Quality Objective Mission: nA
Service: PAX_Call_Service
Service ID: 1
Scenario: PAX_Call_Scenario_ModelA
Scenario ID: 1
Requirement Type: Mixed
Information: Scenario Completed with no errors (Number of validated PAX Call Customizations: 108);

Validation Status: Validated

System: CMS
System ID: 1
Mission: CMSMISSION
Mission ID: 1
Atomic Mission: PLS_AUTO_FSB
Atomic Mission ID: 5
Quality Objective Mission: nA
Service: AUTO_FSB_Service
Service ID: 1
Scenario: AUTO_FSB_Scenario_ModelA
Scenario ID: 1
Requirement Type: Mixed
Information: Scenario Completed with no errors (Flight Phase: Preflight);

Validation Status: Validated

System: CMS
System ID: 1
Mission: CMSMISSION
Mission ID: 1
Atomic Mission: PLS_AUTO_NS
Atomic Mission ID: 6
Quality Objective Mission: nA
Service: AUTO_NS_Service
Service ID: 1
Scenario: AUTO_NS_Scenario_ModelA
Scenario ID: 1
Requirement Type: Mixed
Information: Scenario Completed with no errors (Flight Phase: Preflight);

Validation Status: Validated

System: CMS
System ID: 1
Mission: CMSMISSION
Mission ID: 1
Atomic Mission: PLS_AUTO_FSB
Atomic Mission ID: 5
Quality Objective Mission: nA
Service: AUTO_FSB_Service
Service ID: 1
Scenario: AUTO_FSB_Scenario_ModelA
Scenario ID: 1
Requirement Type: Mixed
Information: Scenario Completed with no errors (Flight Phase: Taxi_Out);

Validation Status: Validated

System: CMS
System ID: 1
Mission: CMSMISSION
Mission ID: 1
Atomic Mission: PLS_AUTO_NS
Atomic Mission ID: 6
Quality Objective Mission: nA
Service: AUTO_NS_Service
Service ID: 1
Scenario: AUTO_NS_Scenario_ModelA
Scenario ID: 1
Requirement Type: Mixed
Information: Scenario Completed with no errors (Flight Phase: Taxi_Out);

Validation Status: Validated

System: CMS
System ID: 1
Mission: CMSMISSION
Mission ID: 1
Atomic Mission: PLS_AUTO_FSB
Atomic Mission ID: 5
Quality Objective Mission: nA
Service: AUTO_FSB_Service
Service ID: 1
Scenario: AUTO_FSB_Scenario_ModelA
Scenario ID: 1
Requirement Type: Mixed
Information: Scenario Completed with no errors (Flight Phase: Take_Off_to80kts);

Validation Status: Validated

System: CMS
System ID: 1
Mission: CMSMISSION
Mission ID: 1
Atomic Mission: PLS_AUTO_NS
Atomic Mission ID: 6
Quality Objective Mission: nA
Service: AUTO_NS_Service
Service ID: 1
Scenario: AUTO_NS_Scenario_ModelA
Scenario ID: 1
Requirement Type: Mixed
Information: Scenario Completed with no errors (Flight Phase: Take_Off_to80kts);

Validation Status: Validated

System: CMS
System ID: 1
Mission: CMSMISSION
Mission ID: 1
Atomic Mission: PLS_AUTO_FSB
Atomic Mission ID: 5
Quality Objective Mission: nA
Service: AUTO_FSB_Service
Service ID: 1
Scenario: AUTO_FSB_Scenario_ModelA
Scenario ID: 1
Requirement Type: Mixed
Information: Scenario Completed with no errors (Flight Phase: Take_Off_V1);

Validation Status: Validated

System: CMS
System ID: 1
Mission: CMSMISSION
Mission ID: 1
Atomic Mission: PLS_AUTO_NS
Atomic Mission ID: 6
Quality Objective Mission: nA
Service: AUTO_NS_Service
Service ID: 1
Scenario: AUTO_NS_Scenario_ModelA
Scenario ID: 1
Requirement Type: Mixed
Information: Scenario Completed with no errors (Flight Phase: Take_Off_V1);

Validation Status: Validated

System: CMS
System ID: 1
Mission: CMSMISSION
Mission ID: 1
Atomic Mission: PLS_AUTO_FSB
Atomic Mission ID: 5
Quality Objective Mission: nA
Service: AUTO_FSB_Service
Service ID: 1
Scenario: AUTO_FSB_Scenario_ModelA
Scenario ID: 1
Requirement Type: Mixed
Information: Scenario Completed with no errors (Flight Phase: Take_Off_to_LiftOff);

Validation Status: Validated

System: CMS
System ID: 1
Mission: CMSMISSION
Mission ID: 1
Atomic Mission: PLS_AUTO_NS
Atomic Mission ID: 6
Quality Objective Mission: nA
Service: AUTO_NS_Service
Service ID: 1
Scenario: AUTO_NS_Scenario_ModelA
Scenario ID: 1
Requirement Type: Mixed
Information: Scenario Completed with no errors (Flight Phase: Take_Off_to_LiftOff);

Validation Status: Validated

System: CMS
System ID: 1
Mission: CMSMISSION
Mission ID: 1
Atomic Mission: PLS_AUTO_FSB
Atomic Mission ID: 5
Quality Objective Mission: nA
Service: AUTO_FSB_Service
Service ID: 1
Scenario: AUTO_FSB_Scenario_ModelA
Scenario ID: 1
Requirement Type: Mixed
Information: Scenario Completed with no errors (Flight Phase: Climb_to400ft);

Validation Status: Validated

System: CMS
System ID: 1
Mission: CMSMISSION
Mission ID: 1
Atomic Mission: PLS_AUTO_NS
Atomic Mission ID: 6
Quality Objective Mission: nA
Service: AUTO_NS_Service
Service ID: 1
Scenario: AUTO_NS_Scenario_ModelA
Scenario ID: 1
Requirement Type: Mixed
Information: Scenario Completed with no errors (Flight Phase: Climb_to400ft);

Validation Status: Validated

System: CMS
System ID: 1
Mission: CMSMISSION
Mission ID: 1
Atomic Mission: PLS_AUTO_FSB
Atomic Mission ID: 5
Quality Objective Mission: nA
Service: AUTO_FSB_Service
Service ID: 1
Scenario: AUTO_FSB_Scenario_ModelA
Scenario ID: 1
Requirement Type: Mixed
Information: Scenario Completed with no errors (Flight Phase: Climb_to1500ft);

Validation Status: Validated

System: CMS
System ID: 1
Mission: CMSMISSION

Mission ID: 1
 Atomic Mission: PLS_AUTO_NS
 Atomic Mission ID: 6
 Quality Objective Mission: nA
 Service: AUTO_NS_Service
 Service ID: 1
 Scenario: AUTO_NS_Scenario_ModelA
 Scenario ID: 1
 Requirement Type: Mixed
 Information: Scenario Completed with no errors (Flight Phase: Climb_to1500ft);

Validation Status: Validated

System: CMS
 System ID: 1
 Mission: CMSMISSION
 Mission ID: 1
 Atomic Mission: PLS_AUTO_FSB
 Atomic Mission ID: 5
 Quality Objective Mission: nA
 Service: AUTO_FSB_Service
 Service ID: 1
 Scenario: AUTO_FSB_Scenario_ModelA
 Scenario ID: 1
 Requirement Type: Mixed
 Information: Scenario Completed with no errors (Flight Phase: Cruise);

Validation Status: Validated

System: CMS
 System ID: 1
 Mission: CMSMISSION
 Mission ID: 1
 Atomic Mission: PLS_AUTO_NS
 Atomic Mission ID: 6
 Quality Objective Mission: nA
 Service: AUTO_NS_Service
 Service ID: 1
 Scenario: AUTO_NS_Scenario_ModelA
 Scenario ID: 1
 Requirement Type: Mixed
 Information: Scenario Completed with no errors (Flight Phase: Cruise);

Validation Status: Validated

System: CMS
 System ID: 1
 Mission: CMSMISSION
 Mission ID: 1
 Atomic Mission: PLS_AUTO_FSB
 Atomic Mission ID: 5
 Quality Objective Mission: nA
 Service: AUTO_FSB_Service
 Service ID: 1
 Scenario: AUTO_FSB_Scenario_ModelA
 Scenario ID: 1
 Requirement Type: Mixed
 Information: Scenario Completed with no errors (Flight Phase: Emergency_Descent);

Validation Status: Validated

System: CMS
 System ID: 1
 Mission: CMSMISSION
 Mission ID: 1
 Atomic Mission: PLS_AUTO_NS
 Atomic Mission ID: 6
 Quality Objective Mission: nA
 Service: AUTO_NS_Service
 Service ID: 1
 Scenario: AUTO_NS_Scenario_ModelA
 Scenario ID: 1
 Requirement Type: Mixed
 Information: Scenario Completed with no errors (Flight Phase: Emergency_Descent);

Validation Status: Validated

System: CMS
 System ID: 1
 Mission: CMSMISSION
 Mission ID: 1
 Atomic Mission: PLS_AUTO_FSB
 Atomic Mission ID: 5

Quality Objective Mission: nA
Service: AUTO_FSB_Service
Service ID: 1
Scenario: AUTO_FSB_Scenario_ModelA
Scenario ID: 1
Requirement Type: Mixed
Information: Scenario Completed with no errors (Flight Phase: Approach);

Validation Status: Validated

System: CMS
System ID: 1
Mission: CMSMISSION
Mission ID: 1
Atomic Mission: PLS_AUTO_NS
Atomic Mission ID: 6
Quality Objective Mission: nA
Service: AUTO_NS_Service
Service ID: 1
Scenario: AUTO_NS_Scenario_ModelA
Scenario ID: 1
Requirement Type: Mixed
Information: Scenario Completed with no errors (Flight Phase: Approach);

Validation Status: Validated

System: CMS
System ID: 1
Mission: CMSMISSION
Mission ID: 1
Atomic Mission: PLS_AUTO_FSB
Atomic Mission ID: 5
Quality Objective Mission: nA
Service: AUTO_FSB_Service
Service ID: 1
Scenario: AUTO_FSB_Scenario_ModelA
Scenario ID: 1
Requirement Type: Mixed
Information: Scenario Completed with no errors (Flight Phase: Landing);

Validation Status: Validated

System: CMS
System ID: 1
Mission: CMSMISSION
Mission ID: 1
Atomic Mission: PLS_AUTO_NS
Atomic Mission ID: 6
Quality Objective Mission: nA
Service: AUTO_NS_Service
Service ID: 1
Scenario: AUTO_NS_Scenario_ModelA
Scenario ID: 1
Requirement Type: Mixed
Information: Scenario Completed with no errors (Flight Phase: Landing);

Validation Status: Validated

System: CMS
System ID: 1
Mission: CMSMISSION
Mission ID: 1
Atomic Mission: PLS_AUTO_FSB
Atomic Mission ID: 5
Quality Objective Mission: nA
Service: AUTO_FSB_Service
Service ID: 1
Scenario: AUTO_FSB_Scenario_ModelA
Scenario ID: 1
Requirement Type: Mixed
Information: Scenario Completed with no errors (Flight Phase: Go_Around);

Validation Status: Validated

System: CMS
System ID: 1
Mission: CMSMISSION
Mission ID: 1
Atomic Mission: PLS_AUTO_NS
Atomic Mission ID: 6
Quality Objective Mission: nA
Service: AUTO_NS_Service
Service ID: 1

Scenario: AUTO_NS_Scenario_ModelA
 Scenario ID: 1
 Requirement Type: Mixed
 Information: Scenario Completed with no errors (Flight Phase: Go_Around);

Validation Status: Validated

System: CMS
 System ID: 1
 Mission: CMSMISSION
 Mission ID: 1
 Atomic Mission: PLS_AUTO_FSB
 Atomic Mission ID: 5
 Quality Objective Mission: nA
 Service: AUTO_FSB_Service
 Service ID: 1
 Scenario: AUTO_FSB_Scenario_ModelA
 Scenario ID: 1
 Requirement Type: Mixed
 Information: Scenario Completed with no errors (Flight Phase: Taxi_In_Start);

Validation Status: Validated

System: CMS
 System ID: 1
 Mission: CMSMISSION
 Mission ID: 1
 Atomic Mission: PLS_AUTO_NS
 Atomic Mission ID: 6
 Quality Objective Mission: nA
 Service: AUTO_NS_Service
 Service ID: 1
 Scenario: AUTO_NS_Scenario_ModelA
 Scenario ID: 1
 Requirement Type: Mixed
 Information: Scenario Completed with no errors (Flight Phase: Taxi_In_Start);

Validation Status: Validated

System: CMS
 System ID: 1
 Mission: CMSMISSION
 Mission ID: 1
 Atomic Mission: PLS_AUTO_FSB
 Atomic Mission ID: 5
 Quality Objective Mission: nA
 Service: AUTO_FSB_Service
 Service ID: 1
 Scenario: AUTO_FSB_Scenario_ModelA
 Scenario ID: 1
 Requirement Type: Mixed
 Information: Scenario Completed with no errors (Flight Phase: Taxi_In_Leave_Runway);

Validation Status: Validated

System: CMS
 System ID: 1
 Mission: CMSMISSION
 Mission ID: 1
 Atomic Mission: PLS_AUTO_NS
 Atomic Mission ID: 6
 Quality Objective Mission: nA
 Service: AUTO_NS_Service
 Service ID: 1
 Scenario: AUTO_NS_Scenario_ModelA
 Scenario ID: 1
 Requirement Type: Mixed
 Information: Scenario Completed with no errors (Flight Phase: Taxi_In_Leave_Runway);

Validation Status: Validated

System: CMS
 System ID: 1
 Mission: CMSMISSION
 Mission ID: 1
 Atomic Mission: PLS_AUTO_FSB
 Atomic Mission ID: 5
 Quality Objective Mission: nA
 Service: AUTO_FSB_Service
 Service ID: 1
 Scenario: AUTO_FSB_Scenario_ModelA
 Scenario ID: 1
 Requirement Type: Mixed

Information: Scenario Completed with no errors (Flight Phase: Post_Flight_AllEnginesOff);

Validation Status: Validated

System: CMS
 System ID: 1
 Mission: CMSMISSION
 Mission ID: 1
 Atomic Mission: PLS_AUTO_NS
 Atomic Mission ID: 6
 Quality Objective Mission: nA
 Service: AUTO_NS_Service
 Service ID: 1
 Scenario: AUTO_NS_Scenario_ModelA
 Scenario ID: 1
 Requirement Type: Mixed
 Information: Scenario Completed with no errors (Flight Phase: Post_Flight_AllEnginesOff);

Validation Status: Validated

System: CMS
 System ID: 1
 Mission: CMSMISSION
 Mission ID: 1
 Atomic Mission: PLS_AUTO_FSB
 Atomic Mission ID: 5
 Quality Objective Mission: nA
 Service: AUTO_FSB_Service
 Service ID: 1
 Scenario: AUTO_FSB_Scenario_ModelA
 Scenario ID: 1
 Requirement Type: Mixed
 Information: Scenario Completed with no errors (Flight Phase: Flight_Phase_FAILURE);

Validation Status: Validated

System: CMS
 System ID: 1
 Mission: CMSMISSION
 Mission ID: 1
 Atomic Mission: PLS_AUTO_NS
 Atomic Mission ID: 6
 Quality Objective Mission: nA
 Service: AUTO_NS_Service
 Service ID: 1
 Scenario: AUTO_NS_Scenario_ModelA
 Scenario ID: 1
 Requirement Type: Mixed
 Information: Scenario Completed with no errors (Flight Phase: Flight_Phase_FAILURE);

Validation Status: Validated

System: CMS
 System ID: 1
 Mission: CMSMISSION
 Mission ID: 1
 Atomic Mission: n/a
 Atomic Mission ID: n/a
 Quality Objective Mission: Power
 Service: n/a
 Service ID: n/a
 Scenario: n/a
 Scenario ID: n/a
 Requirement Type: Non_Functional
 Information: Non-functional parameter value is less or equal than the upper boundary of the objective budget.
 (Overall mission budget is: Root.ObjectivePower,600,2000,7000,0.2);

Power:
 Mean Value: 1690.025445483178
 Lower Boundary: 114.873172044754
 Upper Boundary: 3591.480567604303

Validation Status: Validated

B.5 Generated Validation Report Tabular

The following three tables were taken from the automatically generated validation report for the example of a cabin management system elaborated in chapter 5.

Information
Non-functional parameter value is less or equal than the upper boundary of the objective budget. (Overall mission budget is: Objective{Cost,0,20000,80000,0,4})
Non-functional parameter value is less or equal than the upper boundary of the objective budget. (Overall mission budget is: Objective{Power,600,2000,7000,0,2})
Non-functional parameter value is higher than the lower boundary and less or equal than the mean of the objective budget. (Overall mission budget is: Objective{Weight,0,400,600,0,4})
Scenario Completed with no errors (Audio-Frequency, 8000, 8000, 32000, Audio-Quantization, 8, 8, 32, Volume_Level, 0, 50, 60, Audio-Latency, 0, 6, 32)
Scenario Completed with no errors (Audio-Frequency, 8000, 16000, 32000, Audio-Quantization, 8, 16, 32, Volume_Level, 0, 40, 60, Audio-Latency, 0, 4, 32)
Scenario Completed with no errors (Number of different PA combinations validated: 1677)
Scenario Completed with no errors (Number of validated PAX Call Customizations: 108)
Scenario Completed with no errors (Lavatory Smoke Indication, without CHIME):
Scenario Completed with no errors (Lavatory Smoke Indication, with CHIME, CHIME_TYPE = Type1, VOLUME_dB = 70;
Scenario Completed with no errors (Lavatory Smoke Indication, with CHIME, CHIME_TYPE = Type2, VOLUME_dB = 65;
Scenario Completed with no errors (Lavatory Smoke Indication, with CHIME, CHIME_TYPE = Type3, VOLUME_dB = 80;
Scenario Completed with no errors (Flight Phase: Preflight)
Scenario Completed with no errors (Flight Phase: Taxi.Out)
Scenario Completed with no errors (Flight Phase: Take.Off.to80kts)
Scenario Completed with no errors (Flight Phase: Take.Off.V1)
Scenario Completed with no errors (Flight Phase: Take.Off.to.LiftOff)
Scenario Completed with no errors (Flight Phase: Climb.to400ft)
Scenario Completed with no errors (Flight Phase: Climb.to1500ft)
Scenario Completed with no errors (Flight Phase: Cruise)
Scenario Completed with no errors (Flight Phase: Emergency_Descent)
Scenario Completed with no errors (Flight Phase: Approach)
Scenario Completed with no errors (Flight Phase: Landing)
Scenario Completed with no errors (Flight Phase: Go.Around)
Scenario Completed with no errors (Flight Phase: Taxi.In.Start)
Scenario Completed with no errors (Flight Phase: Taxi.In.Leave.Runway)
Scenario Completed with no errors (Flight Phase: Post.Flight.AllEnginesOff)
Scenario Completed with no errors (Flight Phase: Flight.Phase.FAILURE)
Scenario Completed with no errors (Flight Phase: Preflight)
Scenario Completed with no errors (Flight Phase: Taxi.Out)
Scenario Completed with no errors (Flight Phase: Take.Off.to80kts)
Scenario Completed with no errors (Flight Phase: Take.Off.V1)
Scenario Completed with no errors (Flight Phase: Take.Off.to.LiftOff)
Scenario Completed with no errors (Flight Phase: Climb.to400ft)
Scenario Completed with no errors (Flight Phase: Climb.to1500ft)
Scenario Completed with no errors (Flight Phase: Cruise)
Scenario Completed with no errors (Flight Phase: Emergency_Descent)
Scenario Completed with no errors (Flight Phase: Approach)
Scenario Completed with no errors (Flight Phase: Landing)
Scenario Completed with no errors (Flight Phase: Go.Around)
Scenario Completed with no errors (Flight Phase: Taxi.In.Start)
Scenario Completed with no errors (Flight Phase: Taxi.In.Leave.Runway)
Scenario Completed with no errors (Flight Phase: Post.Flight.AllEnginesOff)
Scenario Completed with no errors (Flight Phase: Flight.Phase.FAILURE)